Computersimulationen verstehen. Ein Toolkit für interdisziplinär Forschende aus den Geistes- und Sozialwissenschaften

Christian Bischof, Nico Formanek, Petra Gehring,
Gabriele Gramelsberger (korrespondierend), Michael Herrmann,
Christoph Hubig, Andreas Kaminski, Felix Wolf

Zur Einleitung

Computersimulationen haben die Wissenschaft verändert und sie spielen auch in anderen gesellschaftlichen Systemen – Politik, Wirtschaft, Massenmedien – eine zunehmend wichtige Rolle. Simulationen verändern den Zukunftsbezug von wissensbasiertem Handeln, indem sie neue Formen politischen Probehandelns, wissenschaftlicher Prognosen, der Gestaltung von Technik oder gesellschaftliche Antizipationen erlauben.

- *Wissenschaft*: Computersimulationen haben die Forschungspraxis und auch die Theoriebildung in den Ingenieurwissenschaften, in Physik, Chemie zugunsten einer methodischen Exploration von zunächst virtuell durchgespielten Möglichkeiten verschoben. Auch in der Medizin sowie in der Soziologie oder Kriminalistik eröffnen Computersimulationen vergleichbare neue Such- und Forschungspfade.
- *Politik und Wirtschaft*: Im politischen System werden Computersimulationen als Chance für ein Probehandeln im gesellschaftlichen Raum gesehen. Man untersucht so beispielweise die Effektivität von erst geplanten Maßnahmen (z.B. Effektivität von Impfstrategien, Beeinflussung ökologischer Kaufentscheidungen, Dynamik gesellschaftlicher Wohlstandsverteilungen) und prüft die Implikationen möglicher politischer Strategien.
- *Massenmedien*: In der Öffentlichkeit haben insbesondere gesellschaftliche Konflikte in Bezug auf Computersimulationen ihren Ort. Werden sie ausgetragen (wie im Fall von Klimasimulationen, aber auch dem simulationsbasierten Flugverbot beim Ausbruch des isländischen Vulkans Eyjafjallajökull 2010), so sind Simulationsergebnisse wie auch die Verfahren, die zu ihnen geführt haben, ein gleichermaßen diskutiertes Thema. Ferner spielen sie in Großforschungsszenarien wie dem mit einer Milliarde Euro geförderten Human Brain Project der EU sowie dort, wo sie sich der Fiktion annähern, etwa in Hollywoodfilmen oder Computerspielen, eine kontroverse Rolle.

Die philosophische, sozialwissenschaftliche oder historische Forschung kommentiert den mit Simulationen verbundenen Methodenwandel unzureichend. Sie hinkt unseres Erachtens dem wachsenden Gewicht von Simulationsforschung hinterher.

Das hat sachliche Gründe, die über eine bloße Scheu vor fremden Fachkulturen oder vor Technik hinausgehen. Die Entwicklung von avancierten Methoden der Computersimulation hat sich in kurzer Zeit erheblich beschleunigt. Es handelt sich dabei um Spezialwissen nicht nur eines Faches, sondern aus verschiedenen Disziplinen, welches in den Simulationsstudien zum Einsatz kommt: Verschiedene Methoden der Reinen und Angewandte

Mathematik, Numerische Algorithmen, parallele Programmierung und entsprechenden Systemarchitekturen sind gleichermaßen wichtig. Die Expertise ist jeweils anspruchsvoll und überschreitet auch die Schreibtischgrenzen der klassischen Wissenschaftstheorie — wie auch der Soziologie und Science and Technology Studies (STS).

In der sogenannten Begleitforschung zu simulationsintensiven Projekten wird daher vielfach improvisiert. Was das erforderliche Wissen angeht, fehlt es, da die universitäre Lehre bislang wenig an Unterstützung anzubieten vermag, an ausgebildeten Akteuren. Wer geistes- und sozialwissenschaftliche Gesprächspartner zum Thema Simulationsforschung sucht, ist auf kontingente biographische Fügungen angewiesen. Auch an Orten, an denen das Thema verfolgt wird, beherrschen Doppelqualifikationen, Mehrfachbegabungen und sehr viel Eigeninitiative das Bild. Was ebenfalls nicht leicht gegeben ist, sind Kontakte zur Praxis der forschenden Informatik, zu Zentren des Hoch- und Höchstleistungsrechnens und zu repräsentativen Forschungsprojekten aus dem natur- und ingenieurswissenschaftlichen Feld. Nicht selten tritt daher der Effekt auf, dass sich geistes- und sozialwissenschaftliche Begleitforscher vor allem mit den besonders frühen (theoretischen) und mit den ganz späten (auf visuelle Ergebnisdarstellung beschränkten) Schritten im Simulationsprozess beschäftigen. Die zahlreichen methodischen Techniken und Kniffe „dazwischen“ bleiben unbekannt und werden nicht berücksichtigt.

Das vorliegende Toolkit, eine Art Hand- oder Überblicksbuch soll Geistes-, Sozial- und Kulturwissenschaftlern die Möglichkeit eröffnen, sich auf breiterer Basis und umfassender mit methodischen Aspekten der Computersimulation zu befassen. Kapitel 1 führt in die insbesondere philosophischen Herausforderungen des Hochleistungsrechnens ein. Kapitel 2 bietet eine Übersicht über die grundlegenden methodischen Zugänge. Kapitel 3 führt dann in die Details einer Simulationsmethode hinein, es widmet sich den gleichungsbasierten Simulationen. Der Grund für diese Art der Vertiefung ist mit dem Zweck des Toolkits verbunden. Die gleichungsbasierte Methode stellt unseres Erachtens die größten mathematischen, numerischen und informatischen Voraussetzungen auf. Sie dürfte für den Erstzugang am schwierigsten sein.

Die Toolkit-Kapitel haben die Absicht, ein Verständnis von Grundprinzipien zu vermitteln – und dabei einen allzu technisch voraussetzungsreichen Jargon zu vermeiden, so dass man in das komplexe Forschungsgebiet auch aus fachfremder Perspektive einsteigen kann. Es geht nicht darum, einen Überblick über die Forschung, Entwicklung und avancierten Problemlagen der verschiedenen Felder zu bieten. Auch einen Vollständigkeitsanspruch aus geistes- und sozialwissenschaftlicher Sicht, hier nämlich: einen vollständigen Eindruck von möglichen Fragestellungen einer geistes- und sozialwissenschaftlichen Begleitforschung zur Simulationsforschung vermitteln zu können, erheben wir nicht.

Zwei Kriterien bestimmten die Auswahl der nachfolgend erläuterten methodischen Schritte: ihr Schwierigkeitsgrad und ihre Relevanz im Feld der Simulationspraxis, die es ja zu beschreiben und verständlich zu machen gilt.

Ob sich die gewählte Herangehensweise als neue Form der Einführung in die methodischen Grundprinzipien der Computersimulation bewähren, wird sich in der Erfahrung erst zeigen müssen. Das Toolkit wird daher in den kommenden Monaten auf der Grundlage der Rückmeldung von Leserinnen und Lesern weiter überarbeitet werden – als Text behält es fürs erste seine digitale Form. Stillschweigende Voraussetzungen sollen weiter expliziert werden, Darstellungsstrategien verändert und auch die Relevanz des einen oder anderen Abschnitts werden vielleicht neu bewertet werden müssen. Hinzukommen sollen außerdem Übungsteile, zudem ist geplant, Angebote zur Schulung von Begleitforschern aus den Geistes- und Sozial- und Kulturwissenschaften zu entwickeln.

Mit Nico Formanek und Michael Herrmann haben wir zwei Wissenschaftler, zwei der oben kurz angesprochenen Doppelbegabungen – ein Physiker und Philosoph sowie ein Mathematiker und Philosoph – ins Projekt einbinden können. Sie haben die Kapitel 2 und 3 verfasst. Ihrer Expertise insbesondere verdanken wir, dass dieses Toolkit erarbeitet worden ist. Das Forum interdisziplinäre Forschung (FiF) der TU Darmstadt war es, das uns die finanziellen Mittel zur Realisierung des im besten Sinne interdisziplinären Vorhabens bereitgestellt hat. Dafür danken wir dem FiF und der Universität Darmstadt.

Petra Gehring, **Christoph Hubig**, **Andreas Kaminski**, **Christian Bischof** und **Felix Wolf**
mit **Nico Formanek** und **Michael Herrmann**

korrespondierend: **Gabriele Gramelsberger**, Universität Witten-Herdecke

Inhaltsverzeichnis

1 Was ist ein Hochleistungsrechner? 7
Petra Gehring, Christoph Hubig, Andreas Kaminski

2 Methoden der Computersimulation und Modellierung 17
Nico Formanek
1 Gleichungsbasierte Simulationen 18
2 Zelluläre Automaten 21
3 Agentenbasierte Simulationen 27
4 Monte-Carlo Simulationen 30

3 Die Simulationspipeline bei gleichungsbasierten Simulationen 35
Michael Herrmann
1 Mathematische Modellierung 36
2 Numerik und Gleitpunktarithmetik 45
3 Implementierung 75
4 Darstellungsfragen/ Visualisierung 80

1 Was ist ein Hochleistungsrechner?

Petra Gehring, Christoph Hubig, Andreas Kaminski

Zwei Perspektiven prägen den Blick auf Hochleistungsrechner und auf das High Performance Computing (HPC). Einerseits sind wir geneigt, solche Verfahren als magisch innovativ zu bewundern: aus unfasslichen Größenordnungen des Prozessierens von Forschungsfragen erwachsen Einsichten neuer Art. Andererseits (und vor allem) erscheinen Hochleistungsrechner als bloße Instrumente. Aufgrund ihrer schieren Rechenkraft können Gegenstände erschlossen werden, die nicht selten sonst unzugänglich blieben, weil keine Experimente mit ihnen möglich sind. Es erscheint dann in keiner Weise aufschlussreich zu ergänzen: Das Instrument, welche sie darstellen, ist ein Recheninstrument.

Doch liegt in der zuletzt genannten Sicht eine Zweideutigkeit. Sofern diese unentdeckt bleibt, verleitet sie zu einer Fehldeutung, welche philosophische und sozialwissenschaftliche Untersuchungen zu diesem Thema auf ungute Weise bremst. Von der Auffassung des Hochleistungsrechners als Recheninstrument scheint die Gedankenfolge gleichsam trivial und daher gänzlich unproblematisch zur Schlussfolgerung zu (ver)führen: Der Höchstleistungsrechner leistet nur einen untergeordneten, ja unwesentlichen Beitrag. Er rechnet ein mathematisches Modellverhalten aus. Alles Entscheidende sowie philosophisch Interessante scheint aus dieser Sicht in den vorherigen Schritten bereits erfolgt zu sein, insbesondere in der fachlichen Modellierung läge so etwas wie das Wesen der Simulation[1].

Die verborgene Zweideutigkeit in diesem vertrauten Bild von der Simulationsforschung kann transparent gemacht werden, indem man sie als Alternative formuliert. Natürlich rechnen Hochleistungsrechner beispielsweise komplexe Gleichungssysteme aus, die zur Lösung von Differentialgleichungssytemen benötigt werden, aber: (i) Rechnen sie diese „nur“ aus oder (ii) tun sie nicht doch, indem sie die Gleichungssysteme lösen, *mehr*?

Die Autorengruppe, die für dieses Buch verantwortlich zeichnet, bekennt sich klar zur Sichtweise (ii). Simulationsverfahren reichen über das bloße Rechnen oder Ausrechnen eines zuvor Modellierten weit hinaus. Mehr ist im Spiel. Um zu verstehen, worin ein solches „Mehr“ bestehen könnte, müssen wir uns den methodischen Schritten in der Durchführung einer Computersimulation zuwenden. Eine verbreitete Idee drückt die Leitmetapher der so genannten Simulationspipeline aus[2]. Demnach wäre für ein zu simulierendes System (den Blutfluss, einen Wirbelsturm, die Ausbreitung einer Krankheit o.Ä) ein beschreibendes)physikalisches)Modell formulierbar. Das Modell wird im nächsten Schritt mathematisch gefasst. Die mathematische Formulierung ist nicht direkt auf den Computer übertragbar, sondern wird mithilfe angewandter Mathematik und Informatik entsprechend umgeformt. Mit Umformen ist im Allgemeinen kein bloßes Übersetzen gemeint, vielmehr können Modelle nur (numerisch) approximiert werden, d.h. man kann sich nicht lückenlos dem theoretisch formulierten Modell annähern und berechnet auf Kosten von Abweichungen und Ungenauigkeiten oft nur eine Nährungslösung, die das Modellverhalten beschreiben

[1] Die Konsequenz ist dann: Die Philosophie der Simulation weist keine eigenständigen, neuartigen Problemlagen auf. Prinzipiell sind ihre Fragen bereits von der Modelltheorie gestellt (und zum Teil) beantwortet worden. Vgl. zu dieser Position Frigg und Reiss 2009.

[2] Vgl. Bungartz u. a. 2013

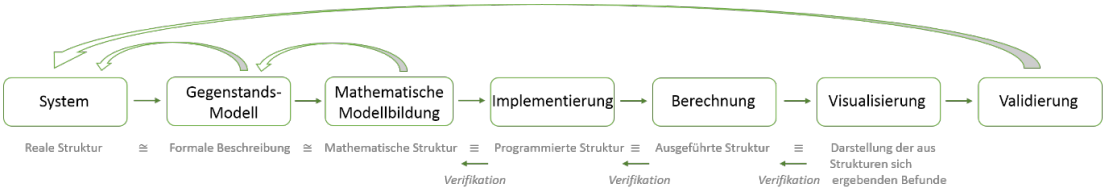


Abbildung 1.1: Detaillierte Simulationspipeline.

soll. Wie „gut“ diese Rechnung auf dem Computer ist, wird noch Gegenstand in diesem Buch sein (vgl. Kapitel 3).

Eine detaillierte Darstellung des so vorgestellten Ablaufs ist Abbildung 1.1 zu entnehmen.

Die Metapher der Pipeline ist durchaus aufschlussreich. Sie unterstellt einen gerichteten Prozess, der von links nach rechts führt. Alles, was links in die Pipeline „gesteckt“ wird, läuft scheinbar bruchlos bweiter nach rechts. Lediglich werden alle Details in die jeweilige Modellierungssprache unter Erhalt der Struktur übersetzt (mathematisch, numerisch, algorithmisch, physischer Rechner). So gesehen läge es tatsächlich nahe, den Hochleistungsrechner als bloßes Recheninstrument zu verstehen. Der große Rechner rechnet – analog zu einem Taschenrechner aus – was manuell aufgrund der vielen Rechenschritte sehr lange dauern würde.

Allerdings bietet die Idee der Simulationspipeline eine unzutreffende Beschreibung des eigentlichen Vorgehens. (1) Es gibt Fälle, in denen es nicht möglich ist, eine „Übersetzung“ von einer Modellierungssprache in eine andere vorzunehmen. Viele komplexe Differentialgleichungssysteme lassen sich analytisch (mit Mitteln der Reinen Mathematik) nicht lösen. Eine Lösung, die man mit Bleistift auf Papier schreiben existiert nicht. Diese Differentialgleichungen müssen notwendigerweise numerisch approximiert werden. Dies bedeutet, dass der Schritt von einem mathematischen Modell auf ein Simulationsmodell (das auf einem Computer dann implementiert werden kann) nicht ohne Verlust an mathematischer Informationen abläuft.
(2) Die methodischen Schritte in der Simulationspipeline sind verschränkt in dem Sinne, dass spätere Schritte den Gestaltungsspielraum früherer Schritte beeinträchtigen können. Beim Schritt vom mathematischen Modell zum Simulationsmodell, man nennt diesen Schritt auch Diskretisierung (genauer wird hierauf in Kapitel 3, Abschnitt 2 muss die Feinheit des Diskretisierungsgitters gewählt werden. Je feiner dieses gewählt wird, desto besser kann eine Lösung auf einem Computer approximiert werden. Allerdings ist das Berechnen einer Näherungslösung auf einem feinem Gitter aufwändiger als auf einem gröberen und erhöht den Hauptspeicherbedarf. Aufwändiger bedeutet hier, dass ein Computer mehr Rechenschritte pro Zeiteinheit machen und mehr Daten abspeichern muss. Dieser Aufwand an Ressourcen stellt ein limitierenden Faktor für die Rechenleistung bei Computern und natürlich beim HPC dar. Somit schränkt die Rechenleistung der verfügbaren Computer die Wahl des Diskretisierungsgitters ein. Dieser Aspekt bei

der Berechnung, sowie die Endlichkeit des Hauptspeichers stellen also von vornherein Einschränkungen an die Genauigkeit der numerischen Approximation, d.h. an die Näherungslösung des Simulationsmodells. Dies hat Folgen. (3) In der Regel werden nämlich mehrere Simulationsläufe durchgeführt, um zu erfahren, wie sich das Simulationsmodell selbst „verhält". Das simulierte Systemverhalten wird dabei mit für bestimmte Bereiche vorliegenden experimentellen Daten verglichen und ggf. angepasst. Es kommt also zu Rückkopplungen, um die Simulationsmodelle zu steuern, indem Parameter im Simulationsmodell verändert oder angepasst werden und die Wirkung im nächsten Simulationslauf „beobachtet"wird. (4) Auch die Ergebnisse eines Simulationsvorganges bedürfen einer Rückübersetzung ins Verständliche. Dies leisten Visualisierungen, deren Rückbezug auf Modelle, Modellierungsprozesse und Modellsprache nicht unbedingt klar ist.

Die Darstellung der Simulationspipeline müsste daher im Grunde sowohl deutlich gekennzeichnete Brüche als auch Rückkopplungspfeile erhalten[3]. Erst dann wäre ein echter Werkzeugkoffer, ein Toolkit des Simulierens gegeben, den auch beobachtende Disziplinen erkennen und verstehen können. Hinzu kommen weitere Kunstgriffe einer Pragmatik der Simulation. Grenzen der Approximation, Anpassungsprozesse und adäquate Darstellungstechniken auch bei sehr schnellen Hochleistungsrechnern führen dazu, dass artifizielle Elemente in das Simulationsmodell eingeführt werden. Bei der Interpretation einer Simulationsstudie muss versucht werden, den Einfluss dieser artifiziellen Elemente auf das Verhalten des Simulationsmodells abzuschätzen. In einem zweiten Schritt kann dann das Verhalten des Simulationsmodells mit den empirischen Daten (oder theoretischen Erwartungen) abgeglichen werden.

Was bedeutet dies alles für die Überlegungen, die uns dazu veranlasst haben, ein Toolkit zu erarbeiten, welches Simulation nicht nur abstrakt schematisiert, sondern in ihrer Pragmatik – und damit auch in ihrer Macht über ihre eigenen Gegenstände – ernst nimmt und darstellt? Wir gingen aus von einer Alternative: (i) Rechnen Hochleistungsrechner ein Modellverhalten „nur" aus oder (ii) tun sie, indem sie etwa Gleichungssysteme lösen, mehr? Die Auffassung, ein Hochleistungsrechner rechnet „nur" komplexe Gleichungssysteme aus, ergab sich aus der mit der ersten Alternative korrespondierenden Idee der unidirektionalen Simulationspipeline. Indem dieses Bild der Lage fraglich wird, werden auch die damit verbundenen, vereinfachenden Auffassungen der Simulationspraxis fraglich und nach Alternativen zu fragen, gewinnt an Plausibilität.

Hochleistungsrechner sind folglich nicht nur ein Recheninstrument. Worin besteht aber das „Mehr"? Die Antwort auf diese Frage möchte unser Toolkit – zumindest ansatzweise – geben. Die Antwort des Toolkit erweitert unsere Vorstellung von der tatsächlichen Bedeutung des Einsatzes von Simulationsverfahren in den experimentierenden, hypothesengetriebenen Naturwissenschaften und auch in der Heuristik des modellbasierten Engineering. Zugleich zeigt sie auf, in welchem Sinne simulationswissenschaftliche Verfahren einschließlich der Darstellung von Ergebnissen (und der zu ihnen führenden Methodik)

[3] Vgl. dazu unter anderem Winsberg 2010, Lenhard und Hasse 2017 und Kaminski u. a. 2016

ein eigener Gegenstand philosophischer, soziologischer oder historischer Forschung sein können.

Grey Box: Hochleistungsrechner als Medium und als Forschungsgegenstand (= „Mehr 1“)

Hochleistungsrechner oder, genauer, Hochleistungsrechner-Umgebungen, denn die Geräte selber sind Teil eines sie durchdringenden und durchformenden operativen Milieus, sind – so eine erste Betrachtungsweise – ein Medium für Forschungsprojekte. Wie andere Medien auch vermitteln Hochleistungsrechner-Umgebungen den Gegenstand, den sie zur Darstellung bringen. Sie „schreiben“ sich in den Gegenstand der Simulationsforschung „ein“, wie eine der gängigen Metaphern lautet, um den Einfluss des jeweiligen Mediums auf seinen Gegenstandsbereich zu benennen.

Hochleistungsrechner sind genau deshalb – philosophisch, aber gerade auch ingenieurwissenschaftlich betrachtet – selbst ein Forschungsobjekt, und zwar eines sui generis. Dies erklärt, warum Hochleistungsrechner nicht nur Service-Personal haben, welches Fachwissenschaftler dabei unterstützt, ihre Simulation „laufen zu lassen“, sondern auch Forscher beteiligt werden müssen, deren Gegenstandsbereich die Funktionsweise des Mediums, also der Hochleistungsrechner selbst ist.

In den Forschungen solcher Wissenschaftler (sog. HPC-Forschung) geht es einerseits darum, die Effekte der jeweiligen Hochleistungsrechenumgebung auf die Simulation, die auf dem Rechner durchgeführt werden soll, ab- und einzuschätzen. Dies ist wesentlich für eine Bewertung der Verlässlichkeit der am Ende erarbeiteten Resultate. Diese Aufgabe der HPC-Forschung ist verwickelter als unter Umständen vermutet wird, wie gleich auch der nächste Punkt – die Frage der sozialen und technischen Organisation des Feldes – deutlich macht[4].

Andererseits muss ein Hochleistungsrechner über seine Lebensdauer hinweg und also auch während der Arbeit in verschiedener Hinsicht optimiert werden. Das führt eine Reihe von technischen Gütekriterien in die Computersimulation ein – wie Robustheit, Effizienz, Performance oder Verlässlichkeit, die aber Teil (weil Voraussetzung) des eigentlichen Rechenvorgangs und damit auch des mittels des Hochleistungsrechners gewonnenen Forschungsergebnisses sind. Dass solche „profan“ erscheinenden Gütekriterien das Forschungsergebnis beeinflussen, ist ein Fingerzeig, der andeutet, warum die technische und damit technikphilosophische – und eben nicht nur die vorherrschende wissenschaftstheoretische – Perspektive auf die Simulation von besonderer Bedeutung ist[5].

[4] Für eine systematische Ausarbeitung von Technik als Medium und zu aktuellen technischen Entwicklungen, welche die Rekonstruktion der Spuren problematisch werden lassen vgl. Christoph Hubig 2006, C. Hubig 2007 und C. Hubig 2015

[5] Es ist spannend zu beobachten, wie in den letzten Jahren eine Art Entdeckung der Technik das philosophische „Vokabular“der Diskussion um Simulationsforschung zu erweitern beginnt. Vorzügliche

Hochleistungsrechner als Forschungsgegenstand sui generis zu betrachten, bedeutet somit letztlich, dass sie auch nicht nur ein Medium zur Erforschung anderer (auf ihrer Grundlage simulierter) Gegenstände sind. Wie sie als Medium funktionieren und wie sie besser funktionieren können sowie welchen Einfluss ihre Funktionsweise auf die Simulationsresultate hat, ist vielmehr ebenfalls zu untersuchen und hat auch eine eigenständig zu erforschende Geschichte, deren Details erst noch erarbeitet werden müssen[6].

Black Box. Die soziale und technische Organisation von HPC (= „Mehr 2“)

Der klassische Wissensbegriff ist an die Rechtfertigung durch eine Person gebunden[7]. Ein Wissensanspruch wird letztlich durch eigene Erfahrung oder eigenes Nachdenken begründet. Die derart individualistische Epistemologie ist freilich durch die soziale Epistemologie in der Nachfolge der Problemlagen der Zeugenschaft unter Druck geraten: In dem, was eine Person zu wissen beansprucht, hängt sie in vielfacher und unter Umständen, je wichtiger auch Geräte im Forschungsprozess werden, in äußerst komplexer Weise vom Wissen anderer Person ab[8]. Paradigmatisch ist dies in der Computersimulation der Fall. Die jeweiligen Fachwissenschaftler (Soziologen, Bau- oder Maschinenbauingenieure, Physiker usw.) müssen mit Mathematikern, mit Numerikern, mit Informatikern usw. zusammenarbeiten und sie setzen eine Welt allen vertrauter, fehlerfrei funktionierender Werkzeuge voraus. Dabei hat man es in allen genannten Bereichen wiederum nicht mit Einzelpersonen zu tun, sondern ganze, durchaus unübersichtliche Gruppen tragen und legitimieren ein Forschungsresultat. In Frage steht so in einer Hochleistungsrechner-Umgebung, in welchem Sinne eine Rechtfertigung für die jeweiligen Simulationsresultate erbracht werden kann. Wie und bis wohin erfolgt eine Rechnernutzung seriös? Rechtfertigungsstrategien, die in der Perspektive der individualistischen Epistemologie verfolgt werden, geraten bei der Diskussion der mit diesem Problem verbundenen Fragen in immense Schwierigkeiten.

Dass die soziale Organisation[9] im Bereich des Hochleistungsrechnens eine besondere Bedeutung hat, könnte zwar bestritten werden. Die jeweiligen Wissenschaftler, könnte man einwenden, müssen gar nicht in einen Austausch miteinander treten; sie verwenden vielmehr gerade in Gestalt von Technik die fertigen Resultate anderer Forschungsbereiche.

Arbeiten zur Philosophie der Simulation liegen unter anderem von Johannes Lenhard vor. Dieser führt dabei eine Reihe von Termini ein, die neuartig für die Wissenschaftsphilosophie sind (Plastizität, Artifizialität, explorative Kooperation). Vgl. insbesondere Lenhard 2015

[6] Vgl. zu dieser u.a. Gramelsberger 2008

[7] Daran ändert die Kritik in der Folge des Gettier-Problems unseres Erachtens nichts. Auch wenn von einer intrinsischen auf die extrinsische Rechtfertigung qua Verlässlichkeit des Verfahrens umgestellt wird, geht es um eine Evidenz, aufgrund derer für eine Person ein Verfahren verlässlich ist, nämlich dank der eigenen Erfahrung.

[8] Vgl. für einen Überblick über die immens angewachsene Diskussion um die soziale Epistemologie und Zeugenschaft Coady 1992

[9] Dies wirft dann auch Fragen der Organisation der so genannten Begleitforschung auf. Vgl. dazu Gehring 2017

Technik wäre damit aus sich heraus ein Garant für Wissenschaftlichkeit. Und der Witz der Technik bestünde darin, ein Wissen von seinem Entstehungs- und Begründungskontext abzulösen, es gleichsam von Rechtfertigungsfragen zu entlasten. Ohne sich um solche zu kümmern, sie zu beachten, Einblick in sie haben zu müssen, könnte ein Rechner also genutzt werden[10]. In der *Entwicklung* der jeweiligen Technik müssten die Begründung für ihre Funktionsweise dann zwar nachvollzogen werden, aber nicht in der *Verwendung* der Technik.

Dieser Einwand verkennt jedoch erstens, dass die Entwicklung von Technik selbst auf der Verwendung anderer Technik basiert. In der Computersimulation zeigt sich das beispielsweise in der Verwendung von Softwarebibliotheken, die nunmehr bei Bedarf „benutzt“ werden. Setzt man aber die Technik modular zusammen, schafft man etwas Neues – eben auch neue etwaige Fehlerquellen und neuen Rechtfertigungsbedarf. Ein zweiter Punkt ist von besonderer Bedeutung für das High Performance Computing: Sehr komplexe Technik ist bereits als Technik nur noch begrenzt in ihrem Funktionieren prüfbar. Hochleistungsrechenumgebungen stellen eine Art epistemischer Undurchschaubarkeit dar. Die einzelnen Komponenten mögen zwar im Rahmen ihrer Fertigung bis ins Detail geprüft und gerechtfertigt sein, das überaus komplexe Zusammenspiel der Komponenten ist dadurch jedoch nicht verstanden, denn es ergibt sich nicht einfach summativ. So kann zwar der Code einer Computersimulation (häufig 100.000 Zeilen) nachvollzogen und geprüft sein; er muss mit dem Compiler zusammenarbeiten (mit Softwarenbibliotheken häufig 2-3 Millionen Zeilen Code). Selbst wenn auch dieser gut verstanden wäre, ist damit nicht geklärt, wie in Quelltext umgeformter Code und Compiler interagieren. In einer epistemischen Ökologie, in welcher überdies die einzelnen Komponenten extrem schnell verschleißen bzw. veralten (die Hardware von Höchstleistungsrechnern wird alle drei bis vier Jahre komplett durch ein neues System ersetzt) und in welcher das Zusammenspiel der Komponenten letztlich insbesondere über längere Erfahrungszeiträume hinweg verstanden werden kann, ist die Rolle der sozialen Verfasstheit sogenannter „Technik“ daher von besonderer Bedeutung. Technik muss hier als Black Box erkannt und behandelt werden, die bei Bedarf nur allmählich ausgeleuchtet werden kann. Die Hochleistungsrechenumgebung als ökologisches System zu behandeln, heißt somit also, mit Grenzen der Nachvollziehbarkeit leben zu lernen. Die Black Box kann nur partiell erhellt, jedoch nicht komplett geöffnet werden[11].

Dark Box. Epistemische Opazität (= „Mehr 3“)

Es gibt eine dritte Ebene, auf welcher Hochleistungsrechner für ein „Mehr“ stehen, das klassische Epistemologien sprengt. Sie betrifft eine für HPC-Forschung erforderliche, sehr spezifische Technisierung der Mathematik. Der Philosoph Paul Humphreys hat

[10] Vgl. zu einer Ausarbeitung dieses Technikbegriffs mit Blick auf die Technisierung der Mathematik Husserl 1976

[11] Weshalb vonseiten der Wissenschaftler, welche Computersimulationen reflektieren, es an einer kohärenten Theorie des HPC mangelt. Vgl. dazu **Resch2017**

diesbezüglich, und zwar mit Blick auf die Frage, ob die Computersimulation neuartige (nicht lediglich neue) Probleme einführt, die These einer epistemischen Opazität – also Undurchsichtigkeit – formuliert[12]. Betroffen ist das Verhältnis von Forscher und Methode. Epistemische Opazität soll heißen, dass Methoden insofern intransparent werden, als man sie nicht mehr vollständig nachvollziehen und rechtfertigten kann – und dies, obwohl sie weiterhin als mathematische Methoden verstanden werden.

Humphreys zufolge existiert eine essentielle, das heißt prinzipiell nicht abbaubare Form der Opazität, die spezifisch ist für das Feld der Computersimulation. Dass es dergleichen nur in der Computersimulation gibt, mag man wissenschaftsphilosophisch bezweifeln: Viele Forschungsbereiche sind sozial und technisch schwer durchschaubar verfasst. Dass eine neuartige Opazität des Einsatzes mathematisierter Methoden im HPC voller Wucht durchschlägt, ist jedoch nachzuvollziehen. Humphreys Diagnose lässt sich sogar vertiefen und prägnanter begründen: Das Phänomen epistemischer Opazität im Hochleistungsrechnen verdankt sich nicht bloß einem Transparenzverlust, sondern einer spezifischen Technisierung der Mathematik[13]. Die Generierung einer Näherungslösung komplexer Differentialgleichungssysteme wird von einem Höchstleistungsrechner erledigt. Unter dem Blickpunkt des praktischen Nachvollzugs können die vielen Prozesse und Rechenschritte auf dem Rechner vom einzelnen Forscher und vom Mathematiker und Informatiker, die für die Modellierung zuständig sind, nicht mehr unmittelbar nachvollzogen werden. Vielfach kann zunächst nur die Modelldynamik beobachtet werden ohne eine theoretische Erklärung aus dem Modell heraus angeben zu können. [14].

Zwei Perspektiven prägen den Blick auf Hochleistungsrechner, hieß es zu Beginn. Einerseits erscheinen sie als Instrumente, als Recheninstrumente. Andererseits führen sie eine neue Methode sowie ein verändertes Verhältnis der Forscher zu ihrer Methode und ihren Resultaten ein. Als Methode betrachtet – und ernst genommen – weist das Hochleistungsrechnen, weisen Simulationsverfahren, einen Überschuss auf. Drei Aspekte haben wir ein wenig spielerisch als „Mehr“ bezeichnet und hervorgehoben: Hochleistungsrechner-Umgebungen sind (technische) Medien, sie sind soziale Milieus oder auch an Öko-Systeme erinnernde komplexe Gefüge und sie verändern – technisieren – den Einsatz von Mathematik. Was in den genannten Bereichen jeweils genau geschieht, wollen – und sollten! – Theoretiker und Praktiker, Ingenieur- und Naturwissenschaftler ebenso wie Geistes- und Sozialwissenschaftler nachvollziehen können.

Die nachfolgenden Kapitel bieten hierzu einen Einstieg an. Sie sollen insbesondere letzteren, also geistes- und sozialwissenschaftlichen Forschungspartnern von ingenieurs- und naturwissenschaftlichen Simulationsforschern einen Einstieg in die raffinierten, aber

[12] Vgl. Humphreys 2004 und Humphreys 2009

[13] Vgl. dazu eine detailliertere Begründung in Kaminski 2013

[14] Vgl. die Folgerung von Lenhard 2017, demzufolge Simulation eine veränderte Mathematik mit sich bringt und in eins damit ein anderes Bild von der Mathematik erforderlich wäre. Lenhard bietet ohnehin das am meisten ausgearbeitete Verständnis für den Zusammenhang Technik und Mathematik in der Computersimulation.

auch verwickelten Methoden eröffnen, die am Hochleistungsrechner, also in der Simulation als Praxis und als wissenschaftlich-technischer Leistung, Verwendung finden.

Literatur

Bungartz, Hans-Joachim u. a. (2013). *Modellbildung und Simulation. Eine anwendungsorientierte Einführung.* Springer Spektrum.

Coady, Cecil Anthony John (1992). *Testimony. A philosophical study.* Oxford University Press.

Frigg, Roman und Julian Reiss (2009). „The philosophy of simulation: hot new issues or same old stew?“ In: *Synthese* 169.3, S. 593–613.

Gehring, Petra (2017). „Doing Research on Simulation Sciences? Questioning Methodologies and Disciplinarities.“ In: *Science and Art of Simulation I (SAS).* Hrsg. von Michael Resch, Andreas Kaminski und Petra Gehring.

Gramelsberger, Gabriele (2008). *Computerexperimente. Zum Wandel der Wissenschaft im Zeitalter des Computers.* Transcript.

Hubig, C. (2007). *Die Kunst des Möglichen II. Grundlinien einer dialektischen Philosophie der Technik.* Transcript.

– (2015). *Die Kunst des Möglichen III. Macht der Technik.* Transcript.

Hubig, Christoph (2006). *Die Kunst des Möglichen I. Technikphilosophie als Reflexion der Medialität.* Transcript.

Humphreys, Paul (2004). *Extending ourselves. Computational science, empiricism, and scientific method.* Oxford University Press.

– (2009). „The philosophical novelty of computer simulation methods“. In: *Synthese* 169.3, S. 615–626.

Husserl, Edmund (1976). *Die Krisis der europäischen Wissenschaften und die transzendentale Phänomenologie. Eine Einleitung in die phänomenologische Philosophie.* Nijhoff (Husserliana IV).

Kaminski, Andreas (2013). „Husserl: Die Krisis der europäischen Wissenschaften und die transzendentale Phänomenologie.“ In: *Nachdenken über Technik. Die Klassiker der Technikphilosophie und neuere Entwicklungen.* Darmstädter Ausgabe. Hrsg. von Christoph Hubig, Alois Huning und Günter Ropohl, S. 186–192.

– (2017). „Der Erfolg der Modellierung und das Ende der Modelle. Epistemische Opazität in der Computersimulation.“ In: *Technik - Macht - Raum. Das Topologische Manifest im Kontext interdisziplinärer Studien.* Hrsg. von Andreas Brenneis u. a.

Kaminski, Andreas u. a. (2016). „Simulation als List.“ In: *Technik, List und Tod. Jahrbuch Technikphilosophie 2.* Hrsg. von Gerhard Gamm u. a.

Lenhard, Johannes (2011). „Epistemologie der Iteration. Gedankenexperimente und Simulationsexperimente.“ In: *Deutsche Zeitschrift für Philosophie* 59.1, S. 131–145.

– (2015a). „Kann Technik die Naturgesetze verändern?“ In: *Ding und System. Jahrbuch Technikphilosophie 1.* Hrsg. von G. Gamm u. a.

– (2015b). *Mit allem rechnen - zur Philosophie der Computersimulation.* de Gruyter.

Lenhard, Johannes (2017). „The Demon's Fallacy. Simulation Modeling and a New Style of Reasoning." In: *Science and Art of Simulation I (SAS)*. Hrsg. von Andreas Resch M.and Kaminski und P. Gehring.

Lenhard, Johannes und Hans Hasse (2017). „Fluch und Segen: die Rolle anpassbarer Parameter in Simulationsmodellen." In: *Jahrbuch Technikphilosophie 3*. Hrsg. von P. Gamm G.and Gehring, A. Hubig C.and Kaminski und A. Nordmann.

Resch, M. (2017). „On the missing coherent theory of simulation". In: *Science and Art of Simulation I (SAS)*.

Winsberg, Eric B. (2010). *Science in the age of computer simulation*. University of Chicago Press.

2 Methoden der Computersimulation und Modellierung

Nico Formanek

1 Gleichungsbasierte Simulationen

1.1 Definition

Gleichungsbasierte Simulationen lösen Gleichungen. Die simulierten Gleichungen sind typischerweise Differentialgleichungen, können aber auch Matrix- oder Funktionsgleichungen sein.

1.2 Grundlagen

Differentialgleichungen sind für reelle Zahlen, die ein Kontinuum bilden, definiert. Da Computer diskrete Systeme sind und nur mit natürlichen Zahlen umgehen können, müssen diese Gleichungen in Differenzengleichungen umgewandelt werden. Hierbei entstehen unvermeidbar Diskretisierungsfehler (siehe Abschnitt 2). Die Eingabe einer gleichungsbasierten Simulation sind die Anfangszustände des Systems, ausgegeben werden eine Vielzahl von Punkten, die die Lösung der Differentialgleichung approximieren.

1.3 Typische Anwendungsfelder

Da Differentialgleichungen in der Physik eine große Rolle spielen, finden gleichungsbasierte Simulationen hauptsächlich dort ihre Anwendung: Um einen Teilchenbeschleuniger zu bauen, muss man das elektromagnetische Feld im Inneren der Beschleunigerstruktur kennen. Da die komplizierten Geometrien keine analytischen Lösungen der Maxwellgleichungen zulassen, simuliert man die Feldverteilung und passt anhand der gewünschten Ergebnisse z.B. die Form der Magneten an.

Die Entstehung von Galaxien ist ein Prozess der Jahrmillionen von Jahren andauert und aus einer riesigen Anzahl an Teilchen besteht (Cox und Loeb 2008). Auch hier behilft man sich mit gleichungsbasierten Simulationen. Aber die Physik ist nicht das einzige Anwendungsfeld gleichungsbasierter Simulationen. Die Ausbreitung von Krankheiten lässt sich auch mit Differentialgleichungen modellieren und damit simulieren (siehe Kermack und McKendrick 1927).

1.4 Beispiel

Als Beispiel berechnen wir numerische Lösungen zu einer einfachen Differentialgleichung.

$$\frac{dy}{dx} = 2x \tag{2.1}$$

In diesem Fall erkennt man die analytische Lösung $y = x^2$ leicht, was aber allgemein nicht so ist. Wir „simulieren“ diese Gleichung dennoch – das verwendete Verfahren ist

grundlegend (siehe Abschnitt **??**). Zunächst müssen wir die Differentialgleichung in eine Differenzengleichung überführen. Das geht am einfachsten, wenn man sich in Erinnerung ruft, dass die Ableitung einer Funktion an einem Punkt ihrer Steigung an diesem Punkt entspricht. Wir nähern die gesuchte Funktion y durch eine Gerade mit der Steigung $2x$ an. Bewegt man sich ein Stück von dem Ausgangspunkt x_0 weg $x_1 = x_0 + \Delta x$, lautet die genäherte Funktion $y_1 = y(x_0) + \Delta y \approx y(x_0) + 2x_0\Delta x$. Für beliebige Punkte lautet die Differenzengleichung nun:

$$y_{n+1} = y_n + 2x_n\Delta x \tag{2.2}$$

Wir setzen den Startpunkt $x_0 = 0$, $y_0 = 0$ und wählen als Schrittweite $\Delta x = 0,1$. Tabelle 2.1 zeigt die ersten zehn Werte, sowie den numerischen Fehler.

Tabelle 2.1: Tabelle der numerischen Berechnung von y. Die letzte Spalte zeigt die Abweichung der analytischen Lösung $y = x^2$ von der numerischen y_n.

n	x_n	y_n	$y_n - x_n^2$
0	0	0	0
1	0,1	0,02	0,01
2	0,2	0,06	0,02
3	0,3	0,12	0,03
4	0,4	0,2	0,04
5	0,5	0,3	0,05
6	0,6	0,42	0,06
7	0,7	0,56	0,07
8	0,8	0,72	0,08
9	0,9	0,9	0,09
10	1	1,1	0,1

In Abbildung 2.1 sieht man deutlich wie die numerische Lösung immer stärker von der analytischen Lösung abweicht. Dies ist ein Fehler der bei der Diskretisierung entstanden ist. Um ihn zu verkleinern, könnten man die Schrittweite kleiner wählen. Allerdings bedeutet eine kleinere Schrittweite auch immer einen höheren Rechenaufwand, da mehr Differenzengleichungen zu Lösen sind. In der Praxis wird man einen Mittelweg zwischen Rechenleistung und Fehler wählen.

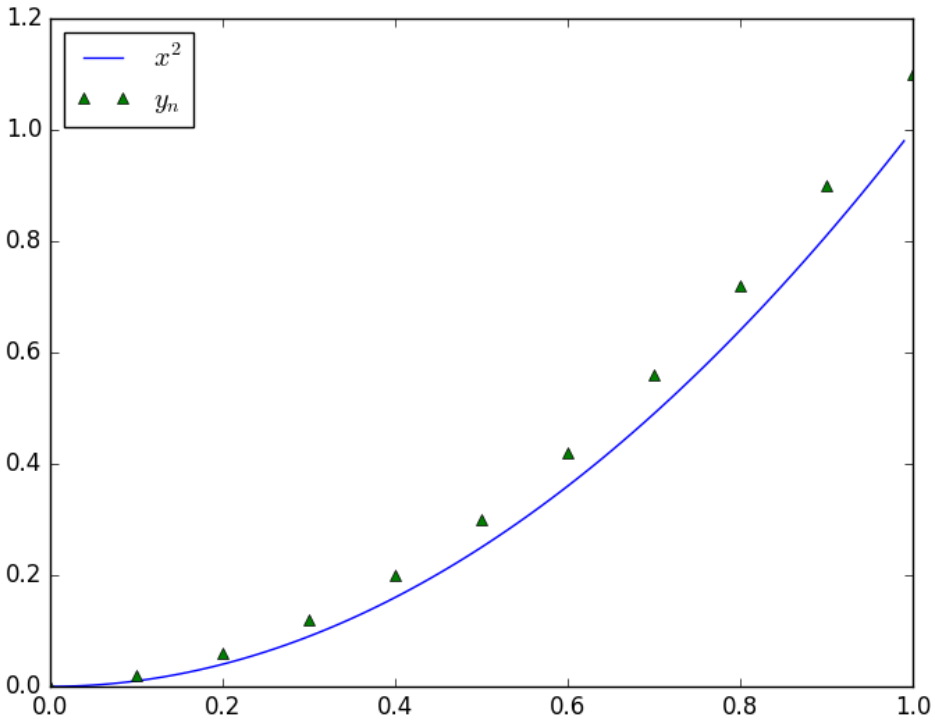


Abbildung 2.1: Verlauf der numerischen x^2 und analytischen y_n Lösung

Literatur

Cox, T. J. und Abraham Loeb (2008). „The collision between the Milky Way and Andromeda“. In: *MNRAS* 386.1, S. 461–474.

Gould, Harvey, Jan Tobochnik und Christian Wolfgang (2006). *An Introduction to Computer Simulation Methods: Applications to Physical Systems*. Addison Wesley.

Kermack, W. O. und A. G. McKendrick (1927). „A Contribution to the Mathematical Theory of Epidemics“. In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 115.772, S. 700.

2 Zelluläre Automaten

2.1 Definition

Es gibt verschiedene Definitionen von zellulären Automaten. Eine gängige Definition aus der Stanford Encyclopedia of Philosophy (vgl. Berto und Tagliabue 2012) lautet:

Definition (Zellulärer Automat).

- *Ein zellulärer Automat besteht aus einem n-dimensionalen Gitter. Die Gitterpunkte werden auch als Zellen bezeichnet.*
- *Er hat nur endlich viele diskrete Zustände. Das heißt die Menge der Zustände Σ ist abzählbar endlich. Ein einzelner Zustand wird üblicherweise mit σ bezeichnet. Die Anzahl der unterscheidbaren Zustände wird mit k bezeichnet.*
- *Für zelluläre Automaten werden nur lokale Wechselwirkungen zugelassen. Üblicherweise hängt der Zustand einer Zelle nur von den nächsten Nachbarn ab. Was als nächster Nachbar zählt, hängt vom Gittertyp ab.*
- *Die Dynamik des zellulären Automats ist diskret und wird durch eine sogenannte Updatefunktion $\phi : \Sigma^n \to \Sigma$ beschrieben. Das Update der Zustände zum Zeitpunkt t geschieht meistens synchron und hängt von den Zuständen zum Zeitpunkt $t-1$ ab.*

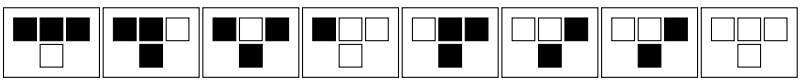
Eine allgemeine Updatefunktion für eindimensionale zelluläre Automaten lässt sich formal wie folgt angeben:

$$\sigma_i(t+1) = \phi(\sigma_{i-r}(t), \sigma_{i-r+1}(t), \ldots, \sigma_{i+r-1}(t), \sigma_{i+r}(t)). \tag{2.3}$$

Der Zustand einer Zelle am Gitterpunkt i zur Zeit t wird hierbei durch $\sigma_i(t)$ angegeben. Der Bereich, aus dem die Nachbarn mit in das Update einbezogen werden, ist r. Für jede Kombination der Nachbarzustände wird ϕ explizit angegeben.

2.2 Grundlagen

Für einen eindimensionalen Zellulären Automaten mit zwei Zuständen ($k = 2$) und einem linken und rechten Nachbarn ($r = 1$) kann ϕ graphisch wie folgt angegeben werden:



0 1 1 0 1 1 1 0

Die zwei Zustände sind schwarz und weiß. Die obere Zeile beschreibt den Zustand zum Zeitpunkt t, die untere Zeile den resultierenden Zustand zum Zeitpunkt $t+1$ abhängig von den Zuständen der zwei nächsten Nachbarn. Das erste große Quadrat bedeutet also

nichts anderes,als dass ein ein schwarzer Zustand, der von zwei schwarzen Nachbarn umgeben ist, im nächsten Zeitschritt weiß wird.

Für zelluläre Automaten mit $k = 2$ und $r = 1$ ist die obere Zeile in der grafischen Darstellung der Regeln gleich, da sie alle Kombinationsmöglichkeiten beinhaltet. Dies ermöglicht eine noch kompaktere Darstellung. Zunächst identifiziert man den Zustand *schwarz* mit der binären 1 und den Zustand *weiß* mit der binären 0. Nun gibt man die Abfolge von schwarz und weiß in der zweiten Zeile als binäre Zeichenkette an. In diesem Fall ist das 01101110, was in Dezimaldarstellung 110 ist. Damit ist der Name Regel 110 motiviert. Diese Art der Bezeichnung von Regeln (auch Wolfram Code genannt) hat sich für die eindimensionalen zellulären Automaten durchgesetzt und wird in der Literatur eingängig verwendet.

Wählt man als Ausgangszustand eine einzige schwarze Zelle, ergibt sich die in Abbildung 2.2 gezeigte Zeitevolution.

Betrachtet man Abbildung 2.2, so kann man Bereiche mit regelmäßigen und eher zufälligen Mustern ausmachen. Die genaue Verteilung der Muster ist extrem stark von dem Anfangszustand abhängig, in dem der Automat gestartet wurde, wie Abbildung 2.3 zeigt.

2.3 Typische Anwendungsfelder

Die Anwendungen von zellulären Automaten sind vielfältig, beispielhaft seien deshalb nur drei Anwendungen genannt.

Physik

Zelluläre Automaten eigenen sich zu Simulation von Gasen und Flüssigkeiten. Die sogenannten Gitter-Gas-Automaten haben Zustände, die verschiedenen Teilchen mit verschiedenen Geschwindigkeiten entsprechen. Als Topologie wird meist ein hexagonales Gitter gewählt. Die Regeln simulieren Teilchenpropagation, sowie Kollision mit Impulserhaltung und sind reversibel. Wird das Gitter entsprechend groß gewählt, so beobachtet man Turbulenzeffekte, wie sie auch von der Navier-Stokes Gleichung beschrieben werden. Da die Gitter-Gas Automaten reversibel und deterministisch sind, kann man Turbulenzen extrem effizient berechnen.

Biologie

Obwohl die Muster auf bestimmten Kegelschneckenarten durch komplexe neurosekretorische Prozesse hervorgerufen werden, lassen sie sich durch einfache zelluläre Automaten reproduzieren. Der Weberkegel (*Conus textile*) zeigt zum Beispiel ein Muster, welches von Regel 30 generiert wird. Abbildung 2.4 zeigt Regel 30 und den Weberkegel im Vergleich.

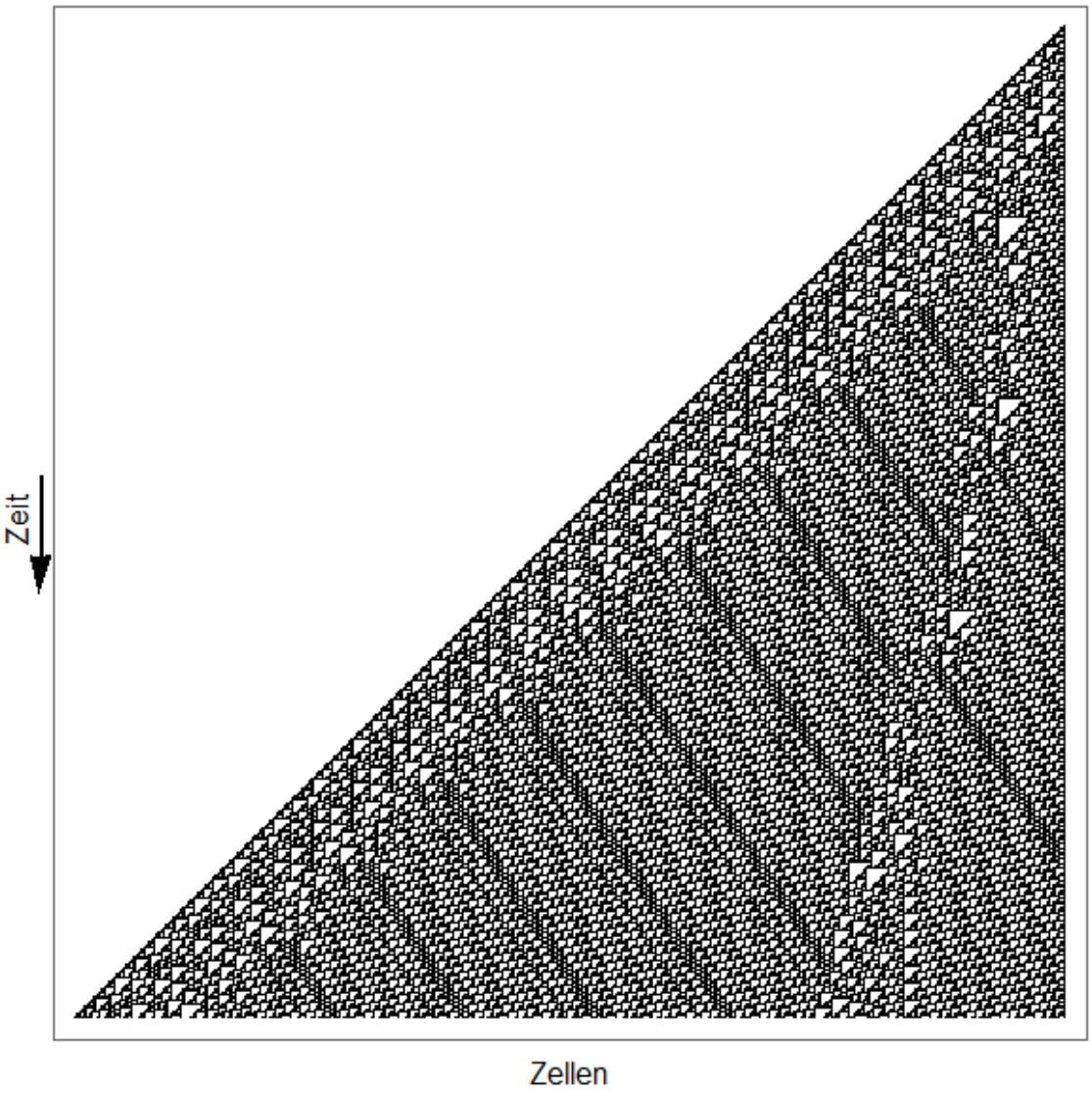
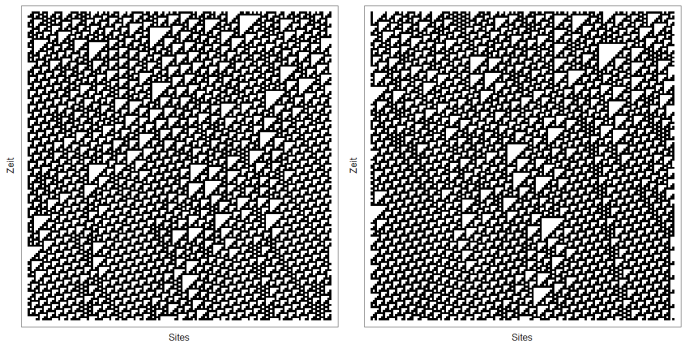


Abbildung 2.2: Zeitevolution einer einzelnen schwarzen Zellen mittels Regel 110

Philosophie

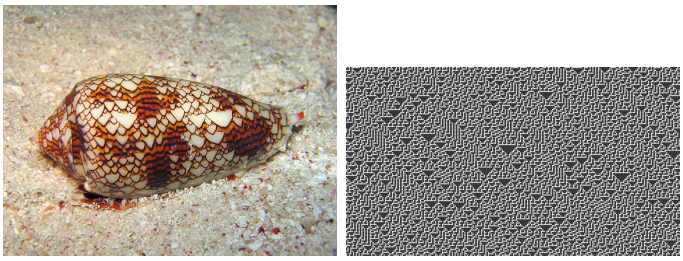
Abgesehen von der Verwendung zum Studium der Emergenz haben zellulären Automaten eine gewisse Popularität als Modell des freien Willens erhalten. Philosophen Dennett, benutzen Beispiele aus dem zellulären Automaten Game of Life, um für eine Position namens Kompatibilismus (d.h. die Vereinbarkeit von Determinismus und freiem Willen) zu argumentieren (vgl. Denett 2004). Die Mikrodynamik von Game of Life ist offensichtlich deterministisch. Auf der Makroebene spricht man von Glidern, Oszillatoren und anderen Strukturen, denen bestimmte Eigenschaften zugeordnet sind. Aus der Makrodynamik



(a) Regel 110 mit einem zufälligen Anfangszustand

(b) Regel 110 mit einem anderen zufälligen Anfangszustand

Abbildung 2.3: Sensitivität von Regel 110 auf den Anfangszustand



(a) Muster des Weberkegels

(b) Von Regel 30 erzeugtes Muster

Abbildung 2.4: Vergleich der Muster von Weberkegel und Regel 30

dieser Strukturen lassen sich viel leichter Vorhersagen über spätere Zustände machen, als durch ausschließliche Betrachtung der Mikrodynamik. Allerdings sind diese Vorhersagen nicht exakt möglich, es sogar gibt Fälle in denen sie falsch sind. Die Analogie mit dem Problem des freien Willens ist nun wie folgt: Eine effektive Beschreibung einer deterministischen Welt macht nur wahrscheinliche und nicht notwendige Aussagen über diese. Obwohl die Mikrowelt determistisch ist, kann es bei fehlender Kenntnis der Dynamik, so scheinen als gäbe es nichtdeterminierte Ereignisse.

2.4 Beispiel

Einer der frühsten und berühmtesten zellulären Automaten heißt Game of Life. Game of Life wird auf einem unendlich ausgedehnten, zweidimensionalen Gitter gespielt. Jede Zelle kann entweder *tot* oder *lebendig* sein und hat acht Nachbarn. Die Updatefunktion ist wie folgt definiert:

1. Eine tote Zelle wird lebendig, wenn sie genau drei lebende Nachbarn hat.
2. Eine lebende Zelle stirbt, wenn sie weniger als zwei lebende Nachbarn hat.
3. Eine lebende Zelle bleibt in ihrem Zustand, wenn zwei oder drei Nachbarn leben.
4. Eine lebende Zelle mit mehr als drei lebenden Nachbarn stirbt.

Aus diesen einfachen und lokalen Regeln ergeben sich viele komplexe Muster[1]. Es existieren Strukturen, die auf dem Gitter propagieren (sogenannte Glider), Oszillatoren, sowie Strukturen die eintreffende Glider vernichten. Mittels dieser und weiterer Strukturen lassen sich Informationen über das Gitter zu übertragen und logische Gatter zu konstruieren. Tatsächlich wurde gezeigt, dass die Berechnungskapazitäten von Game of Life einer universellen Turingmaschine entsprechen (vgl. Berlekamp, Conway und Guy 1982), Game of Life ist also ein universeller Computer. Abbildung 2.5 zeigt eine Momentaufnahme einer Raumschiffkanone, die aus einer Vielzahl von Substrukturen besteht.

[1] Obwohl Game of Life schon 45 Jahre alt ist, werden immer noch neue interessante Strukturen endeckt. Für einen aktuellen Überblick siehe http://www.conwaylife.com/wiki/Main_Page

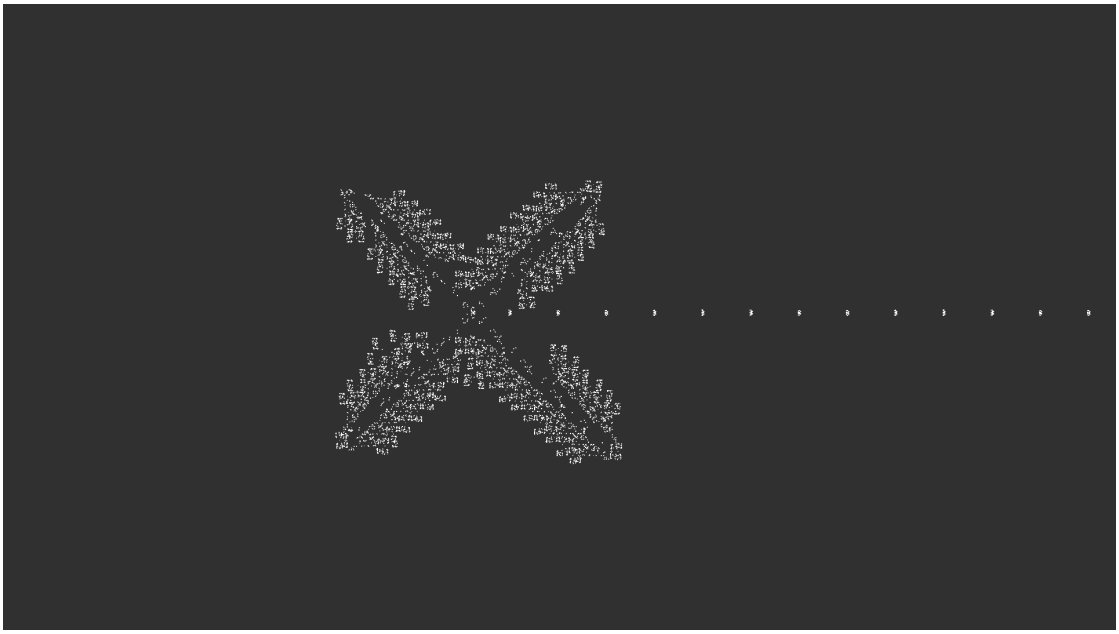


Abbildung 2.5: Eine Raumschiffkanone: Die Raumschiffe werden in der kreuzförmigen Struktur erzeugt und bewegen sich aus der Ebene nach rechts.

Literatur

Berlekamp, E., J. Conway und R. Guy (1982). *Winning Ways for Your Mathematical Plays*. Bd. 2. London: Academic Press.

Berto, Francesco und Jacopo Tagliabue (2012). „Cellular Automata“. In: *The Stanford Encyclopedia of Philosophy*. Hrsg. von Edward N. Zalta. Summer 2012. URL: http://plato.stanford.edu/archives/sum2012/entries/cellular-automata/.

Denett, Daniell (2004). *Freedom Evolves*. Penguin.

3 Agentenbasierte Simulationen

3.1 Definition

Agentenbasierte Simulationen zeichnen sich dadurch aus, dass sie unabhängige Agenten simulieren. Ein Agent ist eine abstrakte Entität, die nach vorgegebenen Regeln handelt und auf ihre Umgebung reagiert. Häufig wird die Umgebung aus anderen Agenten gebildet.

3.2 Grundlagen

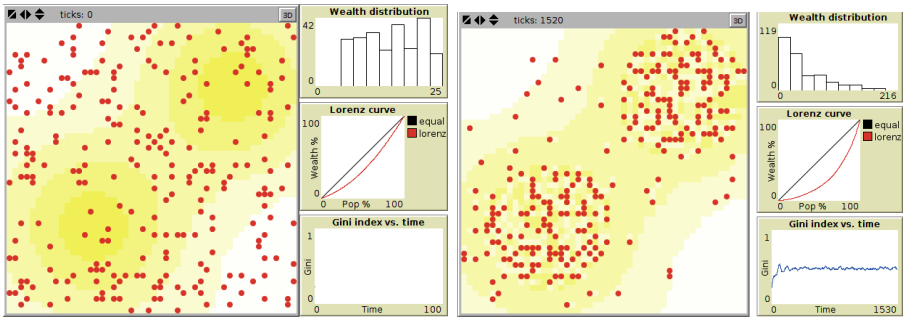
Genau genommen sind agentenbasierte Simulationen eine Unterklasse von zellulären Automaten. Sie werden von diesen unterschieden, da die internen Regeln der Agenten meistens komplexer sind als die der zellulären Automaten (siehe Abschnitt 2). Zudem können verschiedene Agenten mit unterschiedlichen Regeln simuliert werden.
Agentenbasierte Simulationen kommen zum Einsatz, wenn man an der Entstehung komplexen Verhaltens einer Menge von Agenten interessiert ist. Das Ziel der Simulation ist es zu verstehen, wie *globale* Effekte durch die Interaktion *lokaler* Agenten zustandekommen. Dafür muss in der Regel eine Vielzahl von Agenten simuliert werden.

3.3 Typische Anwendungsfelder

Agentenbasierte Simulationen kommen hauptsächlich in den Sozial- und Wirtschaftswissenschaften zum Einsatz. Die erste agentenbasierte Simulation zum Segregationsverhalten von Menschen in Städten, das sogenannte Schellingmodell (siehe Schelling 1969), stammt aus der Soziologie. In der Epidemologie werden agentenbasierte Simulationen zur Untersuchung der Ausbreitung von Seuchen genutzt. Auch in der Verkehrsvorschung finden sie Anwendung, z.B. bei der Vorhersage von Staus. Hierbei werden einzelne Fahrzeuge auf der Autobahn als Agenten simuliert, woraus sich global das Potenzgesetz der Verteilung der Staulängen ergibt.

3.4 Beispiel

Als Beispiel betrachten wir eine Weiterentwicklung von Schellings Segregationssimulation. Die Simulation Sugarscape wurde von Epstein und Axtell (siehe Epstein und Axtell 1996) ersonnen, um die Ungleichheit in der Vermögensverteilung zu analysieren. Statt Geld bildet Zucker das Vermögen. Simuliert wird auf einem zweidimensionalen Gitter mit zwei Zuckergebieten (d.h. eine Ansammlung von Feldern auf denen Zucker wächst – siehe Abbildung 2.6), um die die Zuckerdichte konzentrisch abnimmt. Agenten werden auf einem Feld „geboren“, haben eine bestimmte Sichtweite und ein zufälliges initiales Zuckervermögen. In jedem Schritt der Simulation versucht ein Agent auf das Feld mit der



(a) Startzustand nach zufälliger Verteilung der Agenten auf die Karte.

(b) Zustand nach 1520 Simulationsschritten. Die Agenten haben sich an den Punkten mit viel Zucker angesiedelt. Die Vermögensverteilung ist ungleich geworden.

Abbildung 2.6: Zwei Simulationszustände von Sugarscape (erstellt mit Netlogo (Li und Wilensky 2009)). Rote Punkte sind Agenten, gelbe Felder enthalten Zucker. Je dunkler ein gelbes Feld ist, desto höher ist die Zuckerkonzentration. Die Graphen am rechten Rand ermöglichen einen schnellen Überblick über die erreichte Vermögensungleichheit. Dargestellt ist die kontinuierliche Entwicklung von sozioökonimischen Standardindikatoren wie der Vermögensverteilung, der Lorenzkurve und des Gini-Index.

höchsten Zuckerkonzentration in seiner Sichtweite zu kommen. Falls die aktuelle Position mehr Zucker als alle anderen umliegenden und sichtbaren Felder beinhaltet verbleibt der Agent am Ort. Weiterhin verlieren Agenten in jedem Schritt der Simulation einen Teil ihres Zuckers (Nahrung) und sterben, falls ihr Zuckervermögen auf Null gesunken ist. Außerdem besitzen Agenten eine zufällige Lebenserwartung und sterben, sobald diese erreicht ist. Für jeden verstorbenen Agenten erzeugt die Simulation einen neuen mit zufälligen Startwerten, sodass die Population konstant bleibt. In Abbildung 2.6 sind zwei Zustände von Sucarscape gezeigt.

Literatur

Epstein, Joshua M. und Robert Axtell (1996). *Growing artificial societies: social science from the bottom up*. Brookings Institution Press.

Li, J. und U. Wilensky (2009). *NetLogo Sugarscape 3 Wealth Distribution model*. Center for Connected Learning und Computer-Based Modeling, Northwestern University, Evanston, IL. URL: http://ccl.northwestern.edu/netlogo/models/Sugarscape3WealthDistribution.

Marchi, Scott de und Scott E. Page (2014). „Agent-Based Models“. In: *Annu. Rev. Polit. Sci.* 17, S. 1–20.

Schelling, Thomas C. (1969). „Models of Segregation“. In: *The American Economic Review* 59.2, S. 488–493.

4 Monte-Carlo Simulationen

Der Begriff *Monte-Carlo* bezeichnet diverse Simulationstechniken, die auf Zufallsexperimenten beruhen. Monte-Carlo Simulationen wurden erstmals im Rahmen des Manhattan-Projekts zur Entwicklung von Kernwaffen von Stanislaw Ulam und John von Neumann eingesetzt. Ihre Entwicklung verläuft also parallel zum Aufkommen der ersten Großrechner. Heutzutage sind Monte-Carlo Simulationen in allen Wissenschaften anzutreffen – vor allem die Physik macht starken Gebrauch von ihnen.

4.1 Definition

Es ist schwierig eine Definition anzugeben, die allen Methoden aus dem weiten Bereich der Monte-Carlo Simulation gerecht wird. Gemeinsam ist allen, dass sie den zu simulierenden Prozess mathematisch nicht direkt abbilden. Sie wiederholen vielmehr ein dem Prozess analoges Zufallsexperiment so häufig, bis die Fehlergrenzen im gewünschten Bereich liegen. Bei fast allen Monte-Carlo Methoden verhält sich der Fehler wie

$$\text{Fehler} \approx \frac{1}{\sqrt{\text{Rechenleistung}}}.$$

Dieser Zusammenhang ist eine direkte Konsequenz des zentralen Grenzwertsatzes. Verglichen mit anderen numerischen Methoden (die meistens deutlich spezialisierter auf ein Problem sind), nähern sich Monte-Carlo Methoden nur extrem langsam dem korrekten Wert an. Im Umkehrschluss bedeutet dies, dass man gezwungen ist, mehr Rechenzeit oder -leistung aufzubringen, um verlässliche Ergebnisse zu erhalten. Wegen dieser Eigenschaften kommentiert der Physiker Alan Sokal Monte-Carlo Simulationen wie folgt:

> „Monte Carlo is an extremely **bad** method; it should be used only when all alternative methods are worse."

Dies ist häufig genug der Fall und deshalb sind sie weit verbreitet.

4.2 Grundlagen

Prinzipiell kann man eine Monte-Carlo Simulation „per Hand" mit einigen durch Würfeln erzeugte Zufallszahlen durchführen. Der Computer kommt ins Spiel, wo die durch Würfeln oder natürliche Prozesse erzeugte Anzahl von Zufallszahlen nicht mehr ausreicht. Da der Computer ein determinstisches System ist, kann er keine echte Zufallszahlen erzeugen. Man behilft sich mit Pseudozufallszahlen, die durch mehr oder weniger komplexe Algorithmen vom Computer erzeugt werden. Pseudozufallszahlszahlen sind Reihen von Zahlen, die auf den ersten Blick zufällig aussehen, sich aber nach einer gewissen Zeit wiederholen. Solange diese Periode deutlich größer als die Anzahl der Wiederholungen im

Zufallsexperiment ist, reicht ihre „Zufälligkeit“ aber meist aus. Der Aufbau und Test von Pseudozufallszahlengeneratoren ist eine Wissenschaft für sich und die meisten Forscher nutzen vorgefertigte und geprüfte Algorithmen.

4.3 Typische Anwendungsfelder

Als mathematische Methode dienen Monte-Carlo Simulationen, um hochdimensionale Integrale auszurechnen. Die Grundgleichungen der Physik werden als Differentialgleichungen formuliert, deren Lösungen Integrale sind. Die Anwendungen in der Physik werden sozusagen direkt mitgeliefert. Die Dimension entspricht, z.B. in der Festkörperphysik, nicht mehr der bekannten Raumdimension von $d = 3$, sondern hängt mit der Anzahl der Teilchen im betrachten System (Kristall, Glas, etc.) zusammen. Übliche Freiheitsgrade (d.h. Teilchenanzahlen) und damit Dimensionen sind $d = 10^{23}$. Benötigt ein konventioneller numerischer Algorithmus 10 Stützpunkte, um ein Integral verlässlich zu berechnen, so müssen in 3-Dimensionen $10^3 = 1000$ Werte berechnet werden. Bei 10^{23}-Freiheitsgraden müssten $10^{(10^{23})}$ Werte berechnet werden, deutlich mehr als es Atome im Universum gibt. Monte-Carlo Methoden bieten hier einen Ausweg.
Eine etwas anschaulicheres hochdimensionales Problem ist die Simulation von Galaxien, die von Astrophysikern mit Monte-Carlo Methoden durchgeführt wurde.
Die Anwendungsmöglichkeiten erstrecken sich nicht nur auf die Physik und Mathematik, sondern reichen von der Biologie bis zur Wirtschaftswissenschaft. Eine Zusammenfassung gibt der Übersichtsartikel Kroese u. a. 2014.

4.4 Beispiel

Ein sehr einfaches Beispiel für eine Monte-Carlo Berechung besteht in der Schätzung der Zahl π. Dazu verteilt man zufällig Punkte auf ein Quadrat, dem ein Viertel eines Kreises mit Radius 1 eingeschrieben ist (vgl. Abbildung 2.7), und zählt die Punkte innerhalb des Kreisviertels.

Die Anzahl der Punkte approximiert $\frac{\pi}{4}$, wie man sich recht leicht klarmachen kann: Der Flächeninhalt des Einheitskreises beträgt $A = \pi \times 1^2 = \pi$. Ein Viertel davon ist genau $\frac{\pi}{4}$. Werden die Punkte zufällig auf das Quadrat verteilt, ist die Wahrscheinlichkeit, dass ein Punkt im Kreis landet $\frac{\pi}{4}$. Eine Approximierung dieser Wahrscheinlichkeit ist die relative Häufigkeit $\frac{\text{Punkte im Kreis}}{\text{Gesamtanzahl der Punkte}}$. Damit ist auch leicht ersichtlich, dass Approximationsfehler proportional zur Gesamtanzahl der verteilten Punkte ist. Will man den Fehler senken, muss man die Anzahl der Punkte erhöhen.

Ein Pseudocode des Algorithmus lautet wie folgt:

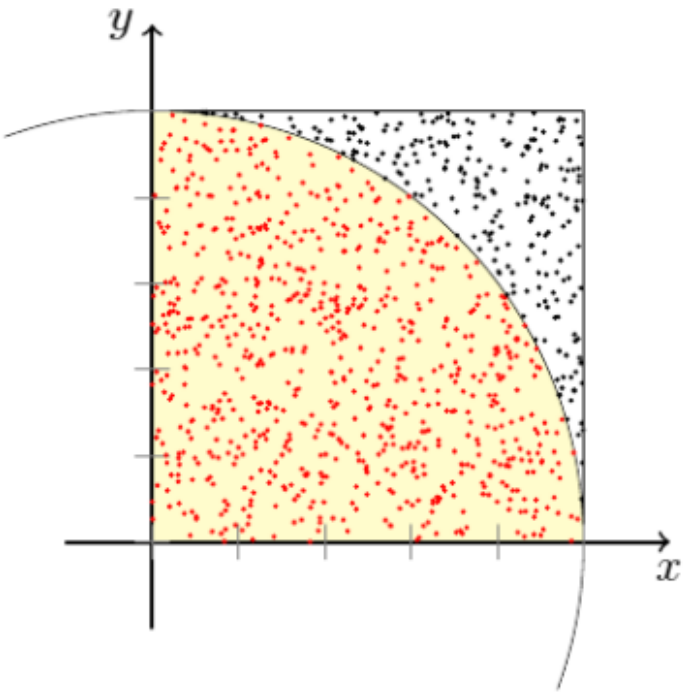


Abbildung 2.7: Zufällige Verteilung der Punkte im Einheitskreis

```
function MCPI(anzahl)
    pi ← 0
    innerhalb ← 0
    gesamt ← anzahl
    while gesamt > 0 do
        x = Random()
        y = Random()
        if x * x + y * y ≤ 1 then
            innerhalb ← innerhalb + 1
        else

    pi ← 4 * innerhalb/anzahl
    return pi
```

Literatur

Anderson, Herbert L (1986). „Metropolis, Monte Carlo, and the MANIAC“. In: *Los Alamos Science* 14, S. 96–108.

Eckhardt, Roger (1987). „Stan ulam, john von neumann, and the monte carlo method“. In: *Los Alamos Science* 15, S. 131–136.

Kalos, Malvin H und Paula A Whitlock (2008). *Monte carlo methods*. John Wiley & Sons.

Kroese, Dirk P u. a. (2014). „Why the Monte Carlo method is so important today“. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 6.6, S. 386–392.

Landau, David P und Kurt Binder (2014). *A guide to Monte Carlo simulations in statistical physics*. Cambridge university press.

Macgillivray, HT und RJ Dodd (1984). „Monte-Carlo simulations of galaxy systems“. In: *Astrophysics and space science* 105.2, S. 331–337.

3 Die Simulationspipeline bei gleichungsbasierten Simulationen

Michael Herrmann

1 Mathematische Modellierung

1.1 Was sind Differentialgleichungen?

Wenn man in der Natur oder in der Technik angesiedelte „Systeme“ zu verstehen versucht, greift man einige als wesentliche und in ihrer Gesamtheit als ausreichend erachtete Zustandsgrößen (wie z.B. Druck, Geschwindigkeit und Lage einzelner Komponenten, Stromstärken an bestimmten Stellen eines Netzwerkes, ...) heraus und überlegt sich, wie diese Größen in ihrer zeitlichen Entwicklung aneinander gekoppelt sind. Das Ergebnis dieser Überlegungen sind die sogenannten konstituierenden Gleichungen des betreffenden Systems. Sie bilden ein mathematisches Modell für das betrachtete System in der Natur oder Technik. Diese Gleichungen sind in aller Regel Differentialgleichungen, die in diesem Abschnitt vorgestellt werden.

Betrachtet man zum Beispiel den Wachstumsprozess von Bakterien in einer Nährflüssigkeit, ist man daran interessiert, zu verstehen, wie sich die Änderungsrate d.h. die Wachstumsgeschwindigkeit, zur Populationsgröße verhält. Sei dazu $P(t)$ die Anzahl der Bakterien zum Zeitpunkt t. Nach Ablauf der kurzen Zeitspanne Δt wird sich sie sich um $\Delta P = P(t + \Delta t) - P(t)$ Mitglieder vermehrt haben. Betrachtet man die steigende Populationsgröße in kleinen Zeitintervallen, kann man die folgende vernünftige Annahme treffen: Bei kleinen Zeitspannen Δt wird die Vermehrung näherungsweise proportional zum Anfangsbestand $P(t)$ und zu der Zeitspanne Δt sein:

$$\Delta P \approx \alpha \cdot P(t) \cdot \Delta t, \tag{3.1}$$

wobei α eine positive Konstante ist, die auch Proportionalitätsfaktor genannt wird und spezifisch für die jeweils betrachtete Bakterienart ist.
Grob gesagt, bedeutet dies: Der Zuwachs ΔP wird sich sowohl bei einer Verdopplung der Population $P(t)$ als auch bei einer Verdopplung der Zeitspanne Δt jeweils verdoppeln.
Wie bereits gesagt, gilt der obige Zusammenhang nur für kleine Δt. Bei großen Δt wird die Proportionalitätsbeziehung unrealistisch, weil die neu hinzukommenden Bakterien ihrerseits ständig zum Wachstum der Population beitragen. Dies wird aber von er obigen Beziehung auch nicht erfasst, da sie nur den zum Zeitpunkt t vorhandenen Bakterien $P(t)$ die Vermehrung in der Zeitspanne Δt erlaubt.

Man schreibt nun (1) in der Form

$$\frac{\Delta P}{\Delta t} \approx \alpha \cdot P(t) \tag{3.2}$$

$\frac{\Delta P}{\Delta t} = \frac{P(t+\Delta t)-P(t)}{\Delta t}$ stellt dabei die Änderungsrate oder Wachstumsgeschwindigkeit dar, mit der sich die Population in der kleinen Zeitspanne Δt ändert. Dieser Ausdruck wird

mathematisch ein Differenzenquotient genannt. Man lässt nun Δt gegen Null streben und kann diesen Ausdruck mit einem Grenzwertausdruck schreiben:

$$P'(t) = \lim_{\Delta t \to 0} \frac{P(t + \Delta t) - P(t)}{\Delta t}$$

$P'(t)$ ist der Betrag, um den sich $P(t)$ für Δt ändert, dividiert durch Δt, wenn Δt gegen Null geht.
Zur gegebenen Funktion $P(t)$ bezeichnet $P'(t)$ die Steigung von $P(t)$ an der Stelle t. Die Funktion $P'(t)$ heißt Ableitung von $P(t)$ und beschreibt hier also das Wachstum der Bakterien.

Damit ist das Wachstumsgesetz für die Bakterienpopulation gefunden:

$$P'(t) = \alpha \cdot P(t)$$

Diese Gleichung beschreibt den Zusammenhang zwischen Ableitung von P, also der Änderungsrate der Population zur Zeit t und der zu dieser Zeit vorhandenen Populationsgröße $P(t)$. Dies stellt eine sogenannte Differentialgleichung dar.

Eine *Differentialgleichung (DGL)* ist eine Gleichung, die eine Funktion f mit deren Ableitungen $f, f'', \ldots$ in Beziehung setzen. Beispiele

- $f'(x) = f(x)$
- $f'(x) = f(x)^2 + \sin x$
- $\sin(f''(x)) = \cos(f'(x))$

Eine Funktion, welche die DGL für alle x erfüllt, heißt *Lösung der DGL*. Mit DGL'en kann man viele Prozesse beschreiben. Die Funktion f beschreibe eine Größe wie:

- den Ort eines Gegenstandes
- die Temperatur, die Konzentration, die Radioaktivität eines Stoffes
- die Größe einer Population (Einwohner, Bakterien, ...)
- den Wert einer Aktie, Währung

Dann beschreibt f' die Änderungsrate dieser Größe, d.h.:

- die Geschwindigkeit des Gegenstandes
- die Zunahme/Abnahme der Temperatur
- das Wachstum der Population

- die Wertsteigerung der Aktie bzw. Währung

f'' beschreibt die Änderungsrate der Geschwindigkeit (Beschleunigung), des Wachstums, usw.

DGL beruhen auf physikalischen Modellen, die der jeweiligen Fragestellung gemäß hinreichend vereinfacht wurden.
So können mit DGL'en u.a. (natur-)wissenschaftliche Gesetze mathematisch formuliert werden, wobei diese Gesetze im Rahmen eines mathematischen Modells aufgestellt werden, das durch vereinfachende Annahmen und durch Abstraktion des eigentlich untersuchten „Systems" gewonnen wird. Das Lösen von Differentialgleichungen ermöglicht Vorhersagen über die Entwicklung der Größen im betrachteten System.

Drei mathematische Fragestellungen sind im Bereich der DGL'en zu bearbeiten:

1. Existenz: Existiert eine Lösung der DGL (überhaupt)?
2. Eindeutigkeit: Ist diese Lösung eindeutig bestimmt, in dem Sinne, dass es für eine gegebene DGL genau eine Lösungsfunktion gibt?
3. Berechenbarkeit: Kann die Lösung (explizit, d.h. mit „Papier und Bleistift") berechnet werden? Wenn nein, wie kann die Lösung näherungsweise berechnet werden? (Numerische Approximation)

Im Folgenden werden diese Fragestellungen am vorigen Beispiel des Bakterienwachstums diskutiert. Die obige Diskussion um die Modellbildung führte auf diese DGL:

$$P'(t) \stackrel{\text{def}}{=} \frac{\partial P(t)}{dt} = \alpha \cdot P(t)$$

Gesucht sind nun Lösungen dieser DGL.
Es sei daran erinnert, dass für $P(t) = e^t$ gilt, dass $P'(t) = e^t$, wobei $e = 2,71828\ldots$. Die Ableitung der Exponentialfunktion ist wieder die Exponentialfunktion, also $P'(t) = P(t)$. Die Exponentialfunktion $P(t) = e^t$ muss nun noch so abgeändert werden, dass bei ihrer Ableitung der Faktor α nach vorne rutscht. Ruft man sich die Kettenregel der Ableitung in Erinnerung, hat man den Ansatz für die Funktion P gefunden: $P(t) = e^{at}$.

Dann ist

$$P'(t) = a \cdot e^{at} = \alpha \cdot P(t)$$

Nun wählt man $a = \alpha$. Somit löst $P(t) = e^{\alpha t}$ die obige DGL.
Man muss jedoch nun beachten, dass für jede reelle Zahl C die Funktion

$$P(t) = C \cdot e^{\alpha t}$$

ebenfalls die DGL löst. Das bedeutet, dass für diese DGL unendlich viele Lösungen existieren (jede Wahl der Konstanten C liefert eine Lösung). In der Regel kennt man aber den Anfangswert der Bakterienpopulation, z.B. $P(0) = 10$. Mit dieser Information lässt sich nun die Konstante C bestimmen:

$$10 = P(0) = C \cdot e^{\alpha 0} = C \cdot 1 = C$$

und somit lautet die nun eindeutig bestimmte Lösung der DGL für das Bakterienwachstum

$$P(t) = 10 \cdot e^{\alpha t}$$

Wie im obigen Fall benötigt man zusätzlich die Angabe von *Anfangs- oder Randbedingungen*[1], die es erlauben, aus der Lösungsmenge die eindeutige Lösung einer gegebenen DGL herauszufinden.

Im Allgemeinen lässt sich die Lösung einer DGL nicht mehr ohne Weiteres explizit von Hand bestimmen (sie muss noch nicht einmal existieren oder eindeutig sein). Die DGL zum Beispiel, die die Auslenkung des mathematischen Pendels beschreibt, lässt sich nur noch näherungsweise lösen (z.B. mit dem Computer, siehe Abschnitt 2).
Die *Ordnung* einer DGL ist durch die höchste auftretende Ableitung festgelegt. So ist die DGL des Bakterienwachstums eine DGL erster Ordnung, weil hier nur die erste Ableitung vorkommt.
DGL'en höherer Ordnung, in der also höhere Ableitungen vorkommen, sind in der Regel schwieriger zu lösen.

1.2 Typen, gewöhnliche und partielle

Manche Prozesse lassen sich dadurch beschreiben und modellieren, dass man das Änderungsverhalten einer Größe/Funktion bezüglich einer einzelnen Variablen betrachtet: typischerweise ist dies meistens die Zeit t. So enthält die DGL im obigen Beispiel die Ableitung nach der Zeit (dies war die Änderungsrate $\frac{dP}{dt}$). Man nennt daher eine DGL, in der zu einer gesuchten Funktion nur Ableitungen nach genau einer Variablen vorkommen, eine *gewöhnliche Differentialgleichung.*

Jedoch lassen sich viele Prozesse nur adäquat beschreiben, wenn das Änderungsverhalten einer Größe/Funktion bezüglich mehrerer voneinander unabhängiger Variablen betrachtet wird. Hängt eine Funktion $u(x, t)$ z.B. vom Ort x und der Zeit t ab, so müssen partielle Ableitungen (also Ableitungen einer Funktion mehrerer Variablen nach einer dieser Variablen) betrachtet werden:

[1] Auf Randbedingungen wird später in Abschnitt 1.3 bei den Navier-Stokes-Gleichungen eingegangen werden.

- $\frac{\partial u(x,t)}{dx}$: Steigung, wenn x variiert wird und t fixiert ist.

- $\frac{\partial u(x,t)}{dt}$: Steigung, wenn t variiert wird und x fixiert ist.

Kommen in einer DGL partielle Ableitungen nach mindestens zwei Variablen vor, spricht man von einer *partiellen Differentialgleichung*. Sie beschreiben komplexe Sachverhalte wie z.B.

1. elektrische und magnetische Felder

2. das Verbiegen von Körpern

3. die Bewegung von Flüssigkeiten

Die Differentialgleichungen werden oft aus physikalischen Prinzipien wie Erhalt von Masse und Impuls und Kräftegleichgewichten hergeleitet. Die Existenz und Eindeutigkeit von Lösungen ist im allgemeinen Fall ein ungelöstes Problem und aktuelles Forschungsgebiet in der Mathematik.

Ein Beispiel ist die sogenannte Transportgleichung:

$$\frac{\partial u(x,t)}{dt} = c \cdot \frac{\partial u(x,t)}{dx}$$

$u(x,t)$ beschreibt z.B. die Konzentration eines Stoffes am Ort x zur Zeit t (Die Konstante c ist für ein spezifisches Problem jeweils vorgegeben).

1.3 Kontinuumsfluiddynamik

Differentialgleichungen beschreiben, wie oben erwähnt, das Änderungsverhalten von Größen. Für viele Anwendungen in der Physik, den Ingenieurswissenschaften und der Klimaforschung ist es interessant zu verstehen, wie sich Fluide, das sind Flüssigkeiten und Gase, im (dreidimensionalen) Raum verhalten.
Strömungen unterschiedlicher Fluide gehören schon lange zu den Phänomenen, die man gerne modellieren und (numerisch) simulieren möchte, da sie in vielen Disziplinen, wie z.B. Astrophysik, Geophysik, Aerodynamik, Klimaforschung, Wettervorhersage oder Medizin studiert werden. Die Simulation von Strömungen ist sehr rechenintensiv und daher oft nur mit Hochleistungsrechnern (High Performance Computing, HPC) umzusetzen.

Mit Hilfe nichtlinearer partieller Differentialgleichungen 2. Ordnung, den sogenannten *Navier-Stokes-Gleichungen*, lässt sich die ganze Vielfalt der Dynamik von newtonschen Fluiden (Fluide mit linear viskosem Fließverhalten) zusammenfassen. Sie beruhen auf der Erhaltung dreier grundlegender Größen in der zugrunde liegenden Wechselwirkung

der Teilchen: Masse, Impuls und Energie. Um die Bewegungen dieser Körper adäquat zu beschreiben, muss man den Druck, die Schwerkraft, die Reibung sowie die Zentrifugal- und Corioliskraft berücksichtigen, die auf eine Flüssigkeit wirken.

Das Strömungsverhalten dynamischer Fluide kann sehr verschieden sein. So können Flüssigkeiten und Gase turbulent werden. Als Turbulenz bezeichnet man eine stark wirbelnde Durchmischung eines Fluides. Diese Wirbel machen die Strömung „kompliziert“. Sie verändert sich mit der Zeit, wird inhomogen, chaotisch und reagiert sehr empfindlich auf Störungen. Kleinste Veränderungen in den Bedingungen können starke Veränderungen in den Strömungen hervorrufen. Diese empfindliche Abhängigkeit von Anfangsbedingungen und Störungen ist für chaotische Systeme typisch (Stichwort: „Schmetterlingseffekt“).

Anschaulich zu bemerken ist die turbulente Bewegung in der Grenzschicht zwischen der Erde und der Atmopshäre. Man nennt diese dort Wind. Bei der Turbulenz ballen sich je nach Größe und Zusammensetzung verschiedene Flüssigkeitsgebiete zu komplizierten Objekten zusammen. Dies verhindert z.B. auch eine langfristige Vorhersage atmosphärischer Strömungen und macht Wettervorhersagen über viele Tage im Voraus unsicher und über Wochen prinizpiell unmöglich.
Turbulente Strömungen und ihr Entstehen sind bis heute theoretisch nicht befriedigend verstanden und entziehen sich einer exakten mathematischen Beschreibung. Mithilfe von Simulationen an Hochleistungsrechnern ist es mittlerweile möglich, turbulente Strömungsmuster mit einfacher Geometrie (zum Beispiel in einem runden, geraden Rohr) zu berechnen.
Ebenso versteht man Strömungen, die kurz davor sind, turbulent zu werden. Langfristig es das Forschungsziel, mathematische Konzepte zu entwickeln, um voll entwickelte Turbulenzen zu beschreiben und zu modellieren. So würde man z.B. die Turbulenz in komplex geformten Fahrzeugen oder in der Atmosphäre besser verstehen. Im Folgenden sollen Turbulenzen vernachlässigt werden. Des Weiteren beschränken wir uns hier auf den Fall einer inkompressiblen Flüssigkeit. Inkompressibel bedeutet, dass sich die Dichte der Flüssigkeit entlang von Teilchenbahnen nicht ändert. Wasser zum Beispiel erfüllt diese Annahme.

Wir erhalten nun das folgende System von Differentialgleichungen, die Navier-Stokes-Gleichungen. Es beschreibt die Geschwindigkeitsverteilung $v(\vec{x}, t)$ in Abhängigkeit vom Ort $\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ zur Zeit t:

$$\operatorname{div} v = 0 \tag{1}$$

$$\rho \left(\frac{\partial v}{\partial t} + (v \cdot \nabla) v \right) = -\nabla p + \mu \Delta v + f \tag{2}$$

Hierbei beschreibt (1) die Divergenzfreiheit des Geschwindigkeitsfeldes. Die Divergenz

div[2], angewendet auf die Geschwindigkeitsfunktion $v(\vec{x}, t)$, misst an jedem Punkt im Raum, wie stark die Geschwindigkeit $v(\vec{x}, t)$ in einer kleinen Umgebung um diesen Punkt auseinander strebt. Für den Fall des Windfeldes in der Atmosphäre quantifiziert die Divergenz das Zusammen- (negative Divergenz) oder Auseinanderströmen (Divergenz) der Luft.

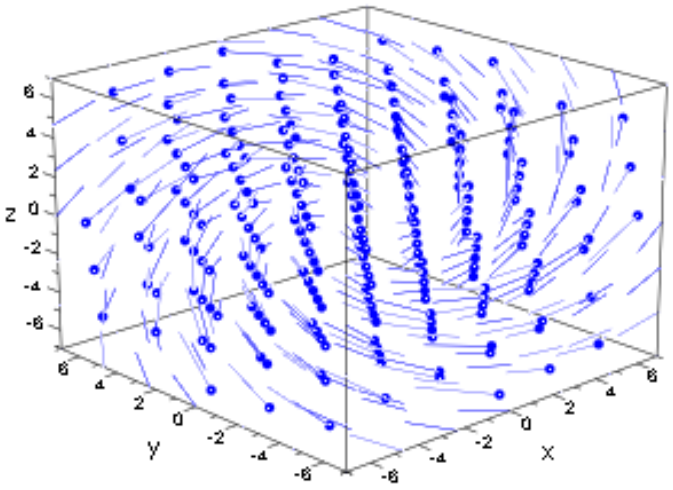


Abbildung 3.1: Darstellung eines Vektorfelds (z.B. das Geschwindigkeitsfeld) mit Divergenz. Diese ist dort positiv, wo die Pfeile auseinanderlaufen, und negativ, wo sie zusammenlaufen.

Im Fall der Divergenzfreiheit $\operatorname{div} v = 0$ ändert sich das Geschwindigkeitsfeld nicht. Anschaulich ausgedrückt gibt es keine Flüssigkeitsquellen bzw. Sickerstellen. Dies stellt eine alternative Charakterisierung inkompressibler Flüssigkeit dar.
In der zweiten Gleichung steht ρ für die Dichte, p für den Druck. Die reelle Zahl μ ist der Viskositätskoeffizient, f sind äußere Kräfte, wie z.B. die Erdanziehung oder Beschleunigungen, denen das System ausgesetzt ist. ∇ und Δ sind mathematische Symbole, die dafür verwendet werden, um alle ersten und zweiten partiellen Ableitungen nach den vorkommenden Variablen Ort $\vec{x}$ und Zeit t zu beschreiben.
Dabei ist ∇ ist der sogenannte Nabla-Operator, der in den Navier-Stokes-Gleichungen u.a. auf die Geschwindigkeitsfunktion $v(\vec{x}, t)$ angewendet wird.
∇v liefert die partiellen Ableitungen nach den Variablen $\vec{x}, t$, aufgelistet in einem Vektor:

$$\nabla v(\vec{x}, t) = \begin{pmatrix} \frac{\partial v}{\partial \vec{x}} \\ \frac{\partial v}{\partial t} \end{pmatrix}$$

Das Symbol Δ bezeichnet den sogenannten Laplace-Operator. Wendet man diesen auf

[2] Die Divergenz, angewendet auf das Geschwindigkeitsfeld $v(\vec{x}, t)$, ist die Summe der partiellen Ableitungen nach den Variablen $\vec{x}$ und t. Formal ausgedrückt heißt dies dann $\operatorname{div} v = \frac{\partial v}{\partial \vec{x}} + \frac{\partial v}{\partial t}$.

die Geschwindigkeitsfunktion $v(\vec{x}, t)$ an, erhält man

$$\Delta v(\vec{x}, t) = \frac{\partial^2 v(\vec{x}, t)}{\partial^2 \vec{x}^2} + \frac{\partial^2 v(\vec{x}, t)}{\partial^2 t^2},$$

das ist gerade die Summe der zweiten partiellen Ableitungen nach dem Ort $\vec{x}$ und der Zeit t.

Für die Erläuterung von Anfangs- und Randbedingungen der Navier-Stokes-Gleichung wird zur Vereinfachung nur der Fall von zwei Raumdimensionen $\vec{x} = (x_1, x_2)$ betrachtet. Ebenso sei angenommen, dass wir ein unverändertes Gebiet vorliegen haben, in dem der Fluss des Fluids stattfindet.
Dieses Gebiet ist durch Wände und eventuell durch Ein- und Ausströmungsöffnungen begrenzt (dies wird hier der Einfachheit halber nicht angenommen). Befindet man sich im Zweidimensionalen, wird das Geschwindigkeitsfeld des Fluids mit $v(\vec{x}, t)$ notiert, welches die Abhängigkeit der Geschwindigkeit von den Dimensionsrichtungen eines zweidimensionalen Szenarios und der Zeit t zeigt.
Zum Zeitpunkt $t = 0$ sind Anfangsbedingungen für das Geschwindigkeitsfeld gegeben, so könnte sich z.B. das Fluid zu Beginn in Ruhe befinden, also setzt man hier $v(\vec{x}, 0)$ für alle Raumpunkte innerhalb des Gebietes.
Die einzelnen Randgebiete des Gebietes legen Bedingungen für die Geschwindigkeit des Fluides fest. Liegen keine Öffnungen des Gebietes vor, müssen keine Ein- oder Ausström-Randbedingungen beachtet werden. Dies führt auf die sogenannten Haftbedingungen („No Slip“):

In diesem Fall wird angenommen, dass das Fluid an den festen, undurchlässigen Wänden haftet und sich somit in deren Nähe im Ruhezustand befindet. Mathematisch bedeutet dies, dass die Geschwindigkeitskomponente senkrecht zum Rand (Normalenrichtung) und die Geschwindigkeitskomponente parallel zum Rand (Tangentialrichtung) auf null gesetzt wird.

Wählt man die Randbedingungen physikalisch richtig (dem gestellten Problem entsprechend), so erhält man ein „gut gestelltes“ Problem, das eine eindeutige Lösung besitzt. Diese Lösung ist jedoch im allgemeinen Fall nicht analytisch berechenbar, d.h. sie kann nicht mithilfe von Formeln auf dem Papier durch dekuktive Schritte in einer geschlossenen Form angegeben werden. Um einer Lösung einer DGL dennoch nahe zu kommen, beginnt man sich über einen Umweg, nämlich mithilfe der Numerischen Mathematik (siehe Abschnitt 2.1), sich einer Approximation an die exakte Lösung anzunähern.

1.4 Grenzen der Modellierung

Zusammen mit der Angabe von Anfangs- und/oder Randbedingungen beschreiben die Gleichungen (1) und (2) inkompressible Flüssigkeiten mathematisch vollständig und stellen somit ein grundlegendes Modell in der Strömungsmechanik dar.

Die mathematisch herausfordernde Aufgabe besteht nun darin, Lösungen für die Navier-Stokes Gleichungen zu finden. Bis heute ist es weder gelungen die Existenz, noch die Eindeutigkeit von globalen Lösungen der Navier-Stokes-Gleichungen analytisch zu beweisen. Es ist also nicht möglich, eine Funktion als Lösung der DGL hinzuschreiben. Zumindest für den Spezialfall der zweidimensionalen inkompressiblen Navier-Stokes-Gleichungen konnte mathematisch bewiesen werden, dass eine Lösung existiert und eindeutig ist.
Für den dreidimensionalen Fall, der für nahezu alle wichtigen Problemstellungen aus der Praxis relevant ist, ist dieser Beweis bis jetzt nicht erbracht.
Dies zeigt die Grenzen der mathematischen Modellierung auf: Zwar ist es gelungen, reale Fragestellungen der Strömungsmechanik in der Sprache der Mathematik auszudrücken, die Komplexität der mathematischen Beschreibung erlaubt uns aber bisher nicht, eine Lösungsfunktion analytisch zu berechnen: Es gibt bis jetzt keine mathematische Theorie mit der es möglich ist, deduktiv die Existenz von Lösungen nachzuweisen.
Dieses Problem ist übrigens eines der sogenannten Milleniums-Probleme, eine Liste von ungelösten Problemen der Mathematik, für die das Clay Mathematics Institute in Cambridge (Massachusetts) ein Preisgeld von einer Million Dollar ausgesetzt hat.
In der Forschung erreicht man Teilfortschritte, indem man die physikalischen Modelle/Randbedingungen vereinfacht und sich Spezialfälle vornimmt. Viele Mathematiker betrachten z.B. den wichtigen Spezialfall der inkompressiblen Navier-Stokes-Gleichung. Hier wurden bereits für den zweidimensionalen Fall Existenz- und Eindeutigkeitssätze bewiesen, für den allgemeinen Fall in drei Raumdimensionen gibt es bislang keinen Existenzbeweis.

Das Folgende dient als Ausblick auf den nächsten Abschnitt, der Antwort darauf gibt, was es für die numerische Mathematik bedeutet, wenn die analytische Lösung einer DGL unbekannt ist.
Während analytische Lösungen der Navier-Stokes-Gleichungen bis jetzt nicht in Reichweite scheinen, wird umso intensiver versucht mit anderen Mitteln, zumindest der exakten, unbekannten Lösung nahe zu kommen: Die Idee ist, die DGL nur noch an endlich vielen Punkten zu betrachten, also aus einem kontinuierlichen ein diskretes Problem zu machen. Dies stellt eine Art „Umweg“ oder Verzerrung dar, welche zwar zu mehr Ungenauigkeit führt, aber zumindest ein Lösungsverfahren ermöglicht, um Approximationen an die unbekannte analytische Lösung tatsächlich zu bestimmen.

Literatur

Bungartz, Hans-Joachim u. a. (2013). *Modellbildung und Simulation: Eine anwendungsorientierte Einführung*. 2., überarb. Aufl. eXamen.press. Berlin: Springer Spektrum. ISBN: 364237655X.

Holzner, Steven und Judith Muhr (2009). *Differentialgleichungen für Dummies: [differenzierter differenzieren geht nicht]*. 1. Aufl. Für Dummies. Weinheim: Wiley. ISBN: 3527705279. URL: http://sub-hh.ciando.com/book/?bok_id=856839.

2 Numerik und Gleitpunktarithmetik

2.1 Warum ist die numerische Modellierung erforderlich?

Die numerische Mathematik, kurz Numerik, ist ein Gebiet der Angewandten Mathematik und befasst sich mit Berechnungsverfahren bzw. sogenannten *Algorithmen* für kontinuierliche mathematische Probleme.
Unter einem Algorithmus versteht man eine Rechenvorschrift, die aus einer endlichen Folge elementarer Rechenschritte besteht und jede dabei auftretende Rechenoperation nach Art und Reihenfolge eindeutig festlegt. Diese Verfahren dienen zum praktischen „Ausrechnen“ von konkreten Zahlenwerten. Das Ausrechnen von (vielen) Zahlenwerten erledigt heutzutage der Computer für uns.

Die numerische Modellierung beruht auf mathematischen Lösungsverfahren für DGL'en. Die dabei betrachteten mathematischen Größen, wie z.B. Geschwindigkeits- und Druckverteilungen oder bestimmte Wachstumsprozesse in der Natur, sind stetige Prozesse. Anschaulich bedeutet dies, dass die entsprechenden Geschwindigkeiten, Druckverteilungen etc. keine „Sprünge“ machen, sondern eine kontinuierliche Ausprägung haben: Alle reellen Zahlen eines Intervalls können mögliche Ausprägungen/Werte dieser mathematischen Größen sein. Ein Merkmal (z.B. die Geschwindigkeit eines Gegenstandes) ist stetig, wenn zwischen zwei Geschwindigkeitswerten immer noch ein weiterer existiert und zwischen diesem und dem vorigen Wert usw. Prinzipiell kann also jeder Zwischenwert als Wert angenommen werden.
Im Gegensatz dazu ist dies bei diskreten Merkmalen nicht der Fall. Diskrete Merkmale besitzen nur endlich oder abzählbar unendlich viele Ausprägungen/Werte. Anschaulich liegen diskrete Zahlen isoliert im Raum, zwischen ihnen gibt es „Löcher“. So ist zum Beispiel die Anzahl von Menschen auf der Erde eine diskrete Zahl. Diskrete Zahlen machen also genau jene „Sprünge“, die die reellen Zahlen nicht tun.

Leider sind der kontinuierliche Formalismus der reinen Mathematik, insbesondere die Sprache der DGL'en und die diskrete Syntax der Computer nicht ineinander überführbar. Denn der mit binärer Kodierung arbeitende Computer kann nur eine endliche, diskrete Menge verschiedener Zahlenwerte darstellen, da es auf dem Computer insbesondere keine reellen Zahlen gibt.
Auf dem Computer stehen für die Darstellung von reellen Zahlen nur endlich viele Stellen zur Verfügung. Zur Approximation von reellen Zahlen und der elementaren arithmetischen Grundoperationen zwischen ihnen werden sogenannte „Maschinenzahlen“ und „Maschinenoperationen“ verwendet, die auf dem Computer realisierbar sind. Alle Eingabedaten müssen daher durch Maschinenzahlen aproximiert werden, bevor man mit ihnen auf dem Computer rechnen kann.

Gleitpunktdarstellung und Gleitpunktarithmetik

Man kann mathematisch beweisen, dass für jede feste natürliche ($b \in \mathbb{N}$) Zahl b, $b > 1$ jede reelle Zahl $x \neq 0$ in der folgenden Form schreiben lässt

$$x = \pm \left(\sum_{i=1}^{\infty} m_i \, b^{-i} \right) \cdot b^e \qquad (\star)$$

wobei der ganzzahlige Exponent e so gewählt werden kann, dass $m_1 \neq 0$ gilt.
Während wir uns vielleicht die Menge der unendlich vielen reellen Zahlen vorstellen können, lassen sich auf einem Computer nur endlich viele Zahlen exakt darstellen. Bei numerischen Berechnungen wird die sogenannte *(normalisierte) Gleitkommadarstellung* (floating point operation) verwendet. Sie ergibt sich, einfach ausgedrückt, aus der oberen Darstellung, indem man nur eine feste Anzahl von Stellen zulässt und den Wertebereich des Exponenten e begrenzt. Diese Darstellung hat dann die zu ($\star$) analoge Form

$$x = m \cdot b^e,$$

wobei

- $b \in \mathbb{N}, b > 1$ die *Basis* des Zahlensystems ist,

- der Exponent e eine ganze Zahl innerhalb gewisser Schranken s, S ist:

$$s \leq e \leq S,$$

- die *Mantisse* m eine feste Anzahl r (die *Mantissenlänge*) an Stellen hat:

$$m = \pm\, 0, m_1, \ldots, m_r, \quad d_j \in \{0, 1, \ldots, b-1\} \text{ für alle } j.$$

Die Eindeutigkeit dieser Darstellung wird dadurch erreicht, indem für $x \neq 0$ gefordert wird, dass $m_1 \neq 0$ ist (Normalisierung, meist wird $m_1 = 1$ gesetzt) (für $x = 0$ setzt man $m = 0$).
Für die Menge der Maschinenzahlen mit Basis b, r-stelliger Mantisse und Exponenten $s \leq e \leq S$ wird die Kurzschreibweise $\mathbb{F}(b, r, s, S)$ verwendet.

Mit dieser Darstellung erhält man somit:

$$\begin{aligned} x &= \pm\, m * b^e \\ &= \pm \left(\sum_{i=1}^{r} m_i\, b^{-i} \right) \cdot b^e \\ &= (m_1 b^{-1} + m_2 b^{-2} + \ldots + m_r b^{-r}) \cdot b^e \end{aligned}$$

Für die Wahl der Basis werden verschiedene Werte für b verwendet. Bei der Wahl von $b = 10$ erhält man das uns bekannte Dezimalsystem. Das Binärsystem mit $b = 2$ ist die mittlerweile auf allen modernen Rechnern übliche Basiswahl, die sowohl technische wie mathematische Vorteile hat.

Beispiel

$$\begin{aligned} 123,75 &= 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-2} \\ &= 2^7 (1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + 0 \cdot 2^{-5} + \\ &\quad 1 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8} + 1 \cdot 2^{-9}) \end{aligned}$$

Diese Zahl kann in einem sechsstelligen Dezimalgleitpunktsystem ($b = 10, r = 6$) dargestellt werden als

$$0,123750 \cdot 10^3$$

In einem 12−stelligen Binärsystem ($b = 2, r = 12$) lautet die gleiche Zahl

$$0,111101111000 \cdot 2^{111}$$

Alle Zahlen, die als Gleitkommazahlen in der obigen Form dargestellt werden können, nennt man auch *Maschinenzahlen*. Da die Anzahl dieser Maschinenzahlen endlich ist, muss es eine betragsmäßig größte und kleineste Maschinenzahl $x_{\min}, x_{\max}$ geben.

$$\begin{aligned} x_{\min} &= (1 * b^{-1} + 0 * b^{-2} + \ldots + 0 * b^{-r}) * b^r = b^{-1} * b^r = b^{r-1} \\ x_{\max} &= (b-1)(b^{-1} + \ldots + b^{-r}) * b^R = (1 - b^{-r}) * b^R \end{aligned}$$

Alle Eingabewerte (reelle Zahlen) in den Computer müssen durch eine endliche Anzahl von Maschinenzahlen ausgedrückt werden. Diese Reduktion leistet eine Rundungsoperation

$$\begin{aligned} \mathrm{rd} : \mathbb{R} &\to \mathbb{F}(b, r, s, S) \\ x &\mapsto \mathrm{rd}(x) \end{aligned}$$

die jeder reellen Zahl im Intervall $[x_{\min}, x_{\max}]$ eine Maschinenzahl zuordnet.
Für jede reelle Zahl $x \in [x_{\min}, x_{\max}]$ mit

$$x = \pm \left(\sum_{i=1}^{\infty} m_i b^{-i} \right) \cdot b^e$$

wird die für die Praxis wichtigste Rundungsart („Korrektes Runden“) so definiert:

$$\mathrm{rd}(x) = \pm \begin{cases} 0, m_1 m_2, \ldots m_r \cdot b^e & \text{falls } m_{r+1} < \frac{b}{2} \\ (0, m_1 m_2, \ldots m_r + b^{-r}) \cdot b^e & \text{falls } m_{r+1} \geq \frac{b}{2} \end{cases}$$

Was hier passiert, ist beispielsweise bei einer 4-stelligen Dezimaldarstellung ($b = 10$) leicht verständlich.
Betrachten wir einmal die beiden Zahlen $0,14283 \cdot 10^0$ und $3,14159 \cdot 10^0$. Man stellt die zu rundende Zahl zuerst in normalisierter Gleitpunktdarstellung dar. Die erste Zahl hat diese Darstellung schon, bei der zweiten lautet sie $0,314159 \cdot 10^1$. Die Rundungsvorschrift sagt nun, dass man hinten nach der vierten Stelle einfach abschneidet, falls die (4+1)-te Ziffer $m_{4+1} < 5$ oder a_4 um 1 erhöht, falls $m_{4+1} \geq 5$ ist. Für unsere beiden Zahlen bedeutet dies also:

$$\begin{aligned} \mathrm{rd}(0,14285 \cdot 10^0) &= 0,1428 \cdot 10^0 \\ \mathrm{rd}(0,314159 \cdot 10^1) &= 0,3142 \cdot 10^1 \end{aligned}$$

Für Zahlen außerhalb des zulässigen, darstellbaren Bereichs $[x_{min}, x_{max}]$ (z.B. wenn die Zahl grösser als x_{max}, kleiner als x_{min} oder das Resultat einer Division durch Null) wird von einigen Maschinen Expontenenüberlauf oder -unterlauf („overflow“, „underflow“registriert. Im ersten Fall wird die Verarbeitung abgebrochen und ein Überlauffehler gemeldet. In den anderen beiden Fällen wird der zu kleine Wert meistens durch Null ersetzt.

Beim Runden macht man zwangsläufig Fehler. Für die Fehleranalyse ist der sogenannte *relative Rundungsfehler*

$$\left| \frac{x - \mathrm{rd}(x)}{x} \right|$$

wichtig (falls $x \neq 0$). Diesen muss man abschätzen, wenn man den möglichen Einfluss von Rundungsfehlern in einem numerischen Algorithmus beurteilen will.

Beispiel

Seien $x = 0,3721448693$ und $y = 0,3720214371$ gegeben. Angenommen der verwendete Computer besitzt 5 Stellen Genauigkeit, d.h. er kann 5-stellige Maschinenzahlen darstellen. Dann gelten $\mathrm{rd}(x) = 0,37214$ und $\mathrm{rd}(y) = 0,37202$. Subtraktion liefert $z = \mathrm{rd}(\mathrm{rd}(x) - \mathrm{rd}(y)) = \mathrm{rd}(0,37214 - 0,37202) = 0,00012$. Das exakte Ergebnis lautet $x - y = 0,0001234322$. Der relative Fehler beträgt also:

$$\frac{|(x-y)-z|}{|(x-y)|} = \frac{0,0000034322}{0,0001234322} \approx 3 \cdot 10^{-2}$$

Dieser Fehler ist sehr groß im Vergleich zu den relativen Fehlern von $\mathrm{rd}(x)$ und $\mathrm{rd}(y)$, die jeweils durch $0,5 \cdot 10^{-4}$ beschränkt sind. Durch Subtraktion zweier in etwa gleich großer Zahlen gingen also 3 Stellen an Genauigkeit verloren!

Man kann zeigen, dass der relative Rundungsfehler nach oben abgeschätzt werden kann:

$$\left|\frac{x - \mathrm{rd}(x)}{x}\right| \leq \frac{1}{2}\, b^{-r+1} = \mathrm{eps}$$

Der Rundungsfehler hängt also im Wesentlichen nur von der Mantissenlänge r und der Basis b ab. Die Schranke

$$\mathrm{eps} = \frac{1}{2}\, b^{-r+1}$$

nennt man *Maschinengenauigkeit*, die von der Computerarchitektur abhängt. Sie gibt also den maximalen Rundungsfehler an und sagt aus, wieviele Ziffern einer reellen Zahl man ungefähr auf dem Computer (hier im Dezimalsystem ausgedrückt) darstellen kann. Heutige Computer arbeiten mit einer Maschinengenauigkeit von $\mathrm{eps} = 2^{-53} \approx 1,1 \cdot 10^{-16}$ (doppelte Genauigkeit[3]). Dies entspricht also einer Genauigkeit auf 16 Stellen im Dezimalsystem.

Durch Umformen der obigen Gleichung folgt direkt,

$$\mathrm{rd}(x) = x(1+\varepsilon) \text{ mit } |\varepsilon| \leq \mathrm{eps}$$

Dieser Rundungsfehler scheint vernachlässigbar klein zu sein, kann aber unter Umständen zu dramatischen Konsequenzen führen, wie das folgende Beispiel zeigt. Im zweiten

[3]Eine Gleitkommazahl mit doppelter Genauigkeit bezeichnet eine Gleitkommadarstellung auf dem Computer, die 8 Byte bzw. 64 Bit belegt. Bei der Darstellung einer Gleitkommazahl werden dabei 11 Bit für den Exponenten, ein Bit für das Vorzeichen und die restlichen 52 Bit für die eigentliche Zahlendarstellung, der Mantisse, verwendet

Golfkrieg sollte eine amerikanische Patriot-Rakete in Saudi-Arabien eine nahende irakische Scud-Rakete zerstören, bevor sie das Ziel, eine amerikanische Kaserne, erreichen würde. Leider verfehlte die amerikanische Rakete ihr Ziel. Die Scud-Rakete schlug in die Kaserne ein und tötete 28 US-Soldaten. Die Ursache war tatsächlich ein Rundungsfehler, der zu einer ungenauen Betrachtung der verstrichenen Zeit seit dem Start des Systems der Patriot-Rakete führte. Die interne Uhr der Patriot-Rakete speicherte die seit dem Hochfahren des Systems verstrichene Zeit in Zehntelsekunden (24 Bit-Register). Da eine Zehntelsekunde im Binärsystem auf dem Computer nicht exakt darstellbar ist, weil sie nicht nach 24 Stellen abbricht (im Binärsystem ergibt sich eine periodisch wiederholende Kommazahl, siehe unten), wurden nur die ersten 24 Stellen verwendet und ein daraus resultierender Rundungsfehler begangen:

$$
\begin{aligned}
0,1\ s &= (0,000\overline{1100})_2\ s \\
&= 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + 1 \cdot 2^{-5} + 0 \cdot 2^{-6} + 0 \cdot 2^{-7} + 1 \cdot 2^{-8} + \ldots\ s \\
&= 0,000\,1100\,1100\,1100\,1100\,1100\,1100\,1100 \ldots\ s \\
&\approx 0,000\,1100\,1100\,1100\,1100\,1100\ s \\
\text{Fehler} &\approx 9,5 \cdot 10^{-8} s
\end{aligned}
$$

Das Patriot-System musste sich in regelmäßigen Abständen neu starten. Nach dem letzten Einschalten war das System nicht heruntergefahren worden. Nach 100 Betriebsstunden akkumulierte sich der Rundungsfehler zu

$$100 * 60 * 60 * 10 * 9,5 * 10^{-8} \text{ Zehntelsekunden } \approx 0,34 \text{ Sekunden.}$$

Die Scud-Rakete bewegte sich mit einer Geschwindigkeit von ungefähr1.676 km/s und war in $0,34$ Sekunden ca. 570 Meter weiter als angenommen. In dieser Zeit legte die iranische Scud-Rakete leider ungefähr 570 Meter zurück und blieb damit außerhalb der Sensoren des Aufspührsystems der Patriot-Rakete. Dies ist umso tragischer, da es aus informatisch-technischer Sicht keinen offensichtlichten Grund gibt in Zehntelsekunden zu messen. Die interne Uhr der Patriot-Rakete hätte die vergangene Zeit genauso gut in 1/8- oder 1/16-Sekunden abspeichern können. Dann wäre eine beliebig lange Betriebsdauer ohne Rundungsfehler möglich gewesen. Allerdings hätte dann sichergestellt werden müssen, dass die Variable für die Anzahl der Zeiteinheiten nicht außerhalb des darstellbaren Bereichs (der benutzten Maschinenzahlen) zu liegen kommt, da es sonst zu einem „overflow“ gekommen wäre. Dies wäre einfach zu bewerkstelligen, indem man das Programm so programmiert, dass es die entsprechende Variable regelmäßig zu einer bestimmten Uhrzeit auf den Wert Null zurücksetzt.

Die exakte Ausführung der arithmetischen Grundoperationen $(+, -, \cdot, /)$ ist mit der Menge der Gleitpunktzahlen im Allgemeinen nicht mehr möglich.

Beispiele:

1. $x = 1,1 \cdot 10^0 = 0,11 \cdot 10^1$ ist eine Gleitpunktzahl zur Basis $b = 10$ mit der

Mantissenlänge $r = 2$, während $x \cdot x = 1,21 \cdot 10^{-1} = 0,121 \cdot 10^{0}$ eine dreistellige Mantisse besitzt.

2. Die Verknüpfung von Maschinenzahlen durch eine exakte elementare arithmetische Operation liefert nicht unbedingt eine Maschinenzahl. Man nehme wieder die Basis $b = 10$ und die Mantissenlänge $r = 3$:

$$0,348 \cdot 10^{2} + 0,785 \cdot 10^{2} = 0,1131 \cdot 10^{3} \neq 0,113 \cdot 10^{3}$$

Die üblichen arithmetischen Operationen $* \in \{+, -, \cdot, /)\}$ müssen also durch geeignete Gleitpunktoperationen $\circledast \in \{\oplus, \ominus, \otimes, \oslash\}$ ersetzt werden, die Maschinenzahlen wieder in Maschinenzahlen überführen (Pseudoarithmetik). Für zwei Maschinenzahlen x, y gilt in der Pseudoarithmetik dann:

$$x \circledast y = \mathrm{rd}(x * y) = (x * y)(1 + \varepsilon),\ |\varepsilon| \leq \mathrm{eps}$$

Grundlegende Regeln der Algebra, die bei exakter Arithmetik gelten, sind in der Pseudoarithmetik nicht mehr gültig. Zum Beispiel spielt auf einmal die Reihenfolge der Verknüpfung eine Rolle (die Assoziativität der Addition geht verloren).

Die numerische Lösung eines Problems besteht in der Regel aus einer Folge von arithmetischer Rechenoperationen bzw. Gleitpunktoperationen. Alle in den Computer eingegebenen (meist reelle) Zahlen werden mit Maschinenzahlen approximiert. In jedem weiteren Schritt wird mit diesen gerundeten Zahlen weitergerechnet. Der Rundungsfehler pflanzt sich im Rechenprozess fort. Bei jedem Algorithmus muss man daher bestrebt sein, das „Aufschaukeln“ von Rundungsfehlern zu vermeiden. Beim Entwerfen von Algorithmen muss man neben dem Faktor Effizienz auch solche Gleitpunktoperationen bevorzugen, die eine möglichst geringe Fehlerakkumulation zur Folge haben. Dieser Aspekt wird als die *Stabilität* eines Algorithmus bezeichnet, was hier aber nicht vertieft wird.
Während die Fehlerquelle der Rundungsfehler grundsätzlich beim Rechnen auf dem Computer auftritt, hat eine zweite Fehlerquelle direkt mit der Formulierung des Algorithmus zu tun: Algorithmen sind in der Regel iterative Verfahren, die in manchen Fällen nicht von alleine abbrechen. In diesen Fällen muss der Algorithmus nach endlich vielen Schritten abgebrochen werden. Mit einem geeigneten Abbruchkriterium wird die Iteration dann angehalten. Somit handelt man sich zusätzlich einen *Abbruchfehler* ein.

Bei der Verarbeitung numerischer Algorithmen treten also zwangsläufig Fehler auf, die durch die Endlichkeit der auf einem Computer darstellbaren Zahlen bedingt sind. Deshalb muss (fast) jede reelle Zahl x durch eine Maschinenzahl $\mathrm{rd}(x)$ gerundet werden. Um mathematische Modelle, die in der Regel auf den reellen Zahlen definiert sind, für den Computer handhabbar zu machen, ist somit notwendigerweise eine numerische Modellierung erforderlich. Dies gilt natürlich insbesondere für DGL'en, bei denen die exakte Lösung analytisch nicht angegeben, also nicht mit „Papier und Bleistift“ gelöst werden können. In solchen Fällen ist man auf numerische Näherungslösungen angewiesen, weil exakte Lösungen grundsätzlich nicht vorhanden sind.

2.2 Was ist die numerische Approximation?

Aufgabenstellung der „numerischen“ Mathematik ist die Entwicklung von Berechnungsverfahren bzw. Algorithmen, mit denen Näherungslösungen mathematischer Problemstellungen effektiv berechnet werden können. Die Näherungslösung ist eine numerische Approximation an die exakte (und oft unbekannte) Lösung eines mathematischen Problems. Um einen Algorithmus angeben zu können, der aus endlich vielen Einzelschritten besteht, ist die Reduktion von der kontinuierlichen Beschreibung (reelle Zahlen) des mathematischen Problems auf eine diskrete Darstellung (Maschinenzahlen) notwendig, wie im obigen Abschnitt erklärt wurde.

Die *Lösung* einer DGL ist kein einzelner Zahlenwert, sondern die Lösung ist selbst eine Funktion, die sozusagen an „unendlich vielen Stützstellen“ unbekannt ist: Für jede reelle Zahl ist der Funktionswert an dieser Stelle nicht bekannt.
Ist nur ein einzelner Zahlenwert unbekannt (ähnlich wie in einer Gleichung mit einer unbekannten Variablen x), spricht man von einem eindimensionalen Problem. Sind zwei Werte nicht bekannt, ist das Problem schon zweidimensional (analog einer Gleichung mit zwei unbekannten Variablen x und y, die schwieriger zu lösen ist als ein eindimensionales Problem). Die Suche nach der Lösungsfunktion einer DGL ist also eine unendlich dimensionale Problemstellung, weil der Funktionswert an jeder Stelle unbekannt ist.
Für die in Abschnitt 1.3 eingeführten Navier-Stokes-Gleichungen beschreibt die Lösungsfunktion die Geschwindigkeitsverteilung $v(\vec{x}, t)$ einer Strömung in Abhängigkeit vom Ort $\vec{x}$ und der Zeit t. Die Angabe der exakten Lösung, d.h. die Angabe des Geschwindigkeitswerts für jeden Ort zu jeder Zeit, kann mit „Papier und Bleistift“ nicht angegeben werden, weil es bis jetzt keine mathematischen Methoden hierfür gibt.
Die exakte Lösung der Navier-Stokes-Gleichungen ist folglich für die Numerik nicht zugänglich. Man versucht daher Näherungslösungen zu finden, indem man das unendlichdimensionale Anfangsproblem (finden der exakten Lösung an allen Stützstellen) durch ein endlichdimensionales Ersatzproblem ersetzt und dieses dann auf einem Computer zu lösen versucht.

2.3 Diskretisierungen, Matrizen, Gitter

Näherungslösungen für ein kontinuierliches Problem durch Lösen eines endlichdimensionalen Ersatzproblems zu gewinnen, wird als *Diskretisierung* bezeichnet. Ein kontinuierliches mathematisches Problem zu diskretisieren, bedeutet eine Rechnung nicht für jede reelle Zahl vorzunehmen, sondern sie nur an endlich vielen Stützstellen auszuführen. Man bezeichnet die Menge dieser Stützstellen auch als *Gitter*.
Für den zweidimensionalen Fall wird anschaulich klar, warum man die Stützstellen als Gitter bezeichnet: Bei den Navier-Stokes-Gleichungen war die Lösung die Geschwindigkeitsfunktion $v(\vec{x}, t)$, die von Ort und Zeit abhing. Wir betrachten hier nun das im Ort $\vec{x}$ diskretisierte Problem, für das man an nur endlich vielen Stützstellen auf der x-Achse die entsprechenden Funktionswerte der Geschwindigkeit berechnet. Trägt man

in einem Koordinatensystem die Stützstellen auf der x-Achse und die entsprechenden Geschwindigkeitswerte auf der y-Achse ab, ergibt sich so natürlicherweise eine Gitterstruktur. Damit ist ein unendlichdimensionales Anfangsproblem in ein endlichdimensionales Ersatzproblem überführt.

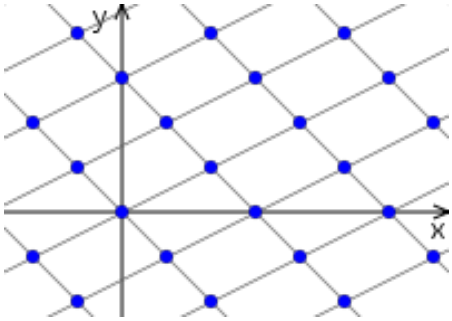


Abbildung 3.2: Berechnet man nur für endlich viele Stützstellen auf der x-Achse die entsprechenden Funktionswerte, ergibt sich eine Gitterstruktur.

Mit der Diskretisierung geht ein *Diskretisierungsfehler* einher, der die Abweichung zwischen der exakten Lösung des Anfangsproblems und der exakten Lösung des Ersatzproblems angibt. Die Summe aller Abweichungen zwischen der exakten Lösung des Anfangsproblems und des numerischen Ergebnisses auf dem Computer wird als *numerischer Fehler* bezeichnet. Er setzt sich zusammen aus Diskretisierungsfehler, Rundungs- und Abbruchfehlern (bei iterativen Verfahren).

Bei allen numerischen Verfahren ist der Begriff der *Stabilität* von großer Wichtigkeit. Ein *stabiles* Verfahren ist ein Verfahren, bei dem kleine Änderungen der Eingabedaten in den Computer nur kleine Änderungen der Ausgabedaten hervorrufen. Das bedeutet, dass sich kleine Abweichungen im Zuge einer Berechnung nicht akkumulieren. So dürfen sich beispielsweise bei einer Diskretisierung Rundungs- und Diskretisierungsfehler nicht aufhäufen, da sonst nach einer hinreichenden Anzahl von Rechenschritten ein nicht zu gebrauchendes Näherungsergebnis erzielt wird.

Ein numerisches Verfahren kann natürlich nicht „stabiler“ sein als das Ausgangsproblem. Man spricht von einem *wohlgestellten Problem*, wenn die exakte Lösung des Anfangsproblems existiert, eindeutig ist und stetig von den Anfangsdaten abhängt. Wenn die exakte Lösung nicht stetig von den Anfangsdaten abhängt, können kleine Änderungen in den Anfangsdaten große Änderungen im Ergebnis bewirken.
Bei allen numerischen Verfahren spielt die *Konvergenz* eine entscheidende Rolle. Man nennt ein Verfahren *konvergent*, wenn die Näherungslösung im Grenzübergang für ein unendlich feines Gitter (Die Anzahl der Stützstellen, an denen der Funktionswert berechnet werden muss, wird immer größer) gegen die exakte Lösung des gestellten Problems strebt. Im Folgenden werden die Begriffe Diskretisierung und Konvergenz am Beispiel der numerischen Integration erläutert.

Gegeben sei eine in einem Intervall $[a, b]$ stetige Funktion f. Gesucht ist das bestimmte Integral

$$\int_a^b f(x)\, dx.$$

Diese Aufgabe ist so auf einem Computer im Allgemeinen nicht zu lösen, da die Integralberechnung für bestimmte komplizierte Funktionen nicht bekannt ist bzw. keine Möglichkeit besteht, das Integral überhaupt analytisch zu lösen. Das Integral ist eine kontinuierliche Größe über den reellen Zahlen und besteht aus unendlich vielen Summanden. Man führt deshalb eine Diskretisierung ein, indem man das Integralzeichen durch eine endliche Summe ersetzt, das Integral sozusagen nur an endlich vielen Stützstellen auswertet.
Da das Integral die Fläche zwischen einer Funktion f und der x-Achse ist, versucht man diese Fläche mithilfe von Rechtecksflächen zu berechnen. Das folgende Schaubild zeigt dies exemplarisch für den Graphen einer beliebigen Funktion f.

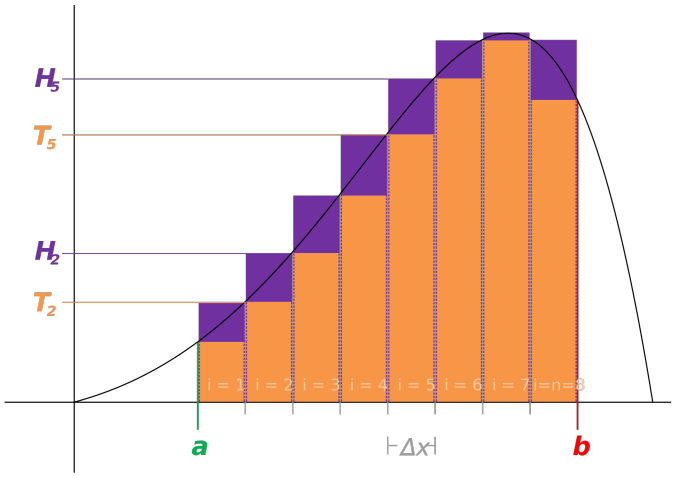


Abbildung 3.3: Hier wird das Intervall $[a, b]$ in $n = 8$ Teilintervalle zerlegt. Die Obersumme ist lila, die Untersumme orange gekennzeichnet.

Die Idee besteht darin, die Fläche zwischen Kurve und x-Achse durch eine Reihe schmaler Rechtecke zu approximieren. Für den exakten Wert eines Integrals lässt man dann die Breite dieser Rechtecke in einem Grenzprozess gegen 0 gehen.
Man fängt also damit an, das Intervall $[a, b]$ in n gleich lange Teilintervalle zu unterteilen (man muss die Stücke nicht gleich lang machen, dies ist aber der Einfachheit halber hier am besten geeignet):

$$\Delta x = \frac{b-a}{n}$$
$$x_i = a + \Delta x \cdot i$$

wobei $x_0 = a, x_n = b$. Im Schaubild wurde das Intervall $[a, b]$ in 8 ($n = 8$) gleich große Teilintervalle zerlegt. Man kann sich nun entscheiden, ob man die Obersumme (diejenigen Rechtecke, die über den Graphen von f hinausragen) oder die Untersumme (diejenigen Rechtecke unterhalb des Graphen von f) für die Approximation des Integrals (Fläche zwischen Graph der Funktion f und der x-Achse) heranzieht. Hier wird die Untersumme gewählt. Die Untersumme verwendet die linke obere Ecke der Rechtecke, um die Höhe des Rechtecks festzulegen. Somit hat das erste Rechteck die Breite Δx und die Höhe $f(x_0)$, was gerade $f(a)$ entspricht, usw. Die Höhe des letzten Rechtecks ist $f(x_{i-1})$. Die Untersumme summiert nun diese Rechtecksflächen auf:

$$U_n = f(a) \cdot \Delta x + f(x_1) \cdot \Delta x + \ldots + f(x_{i-2}) \cdot \Delta x + f(x_{i-1}) \cdot \Delta x$$

Dies lässt sich mit der Summenzeichen ($\sum$) so schreiben:

$$U_n = \sum_{i=1}^{n} f(x_{i-1}) \cdot \Delta x$$

Man nennt n bzw. Δx den *Diskretisierungsparameter*, da er angibt, wie fein man das Intervall in Teilintervalle unterteilt. Aus der Herleitung des Verfahrens ist klar, dass im Grenzübergang für $n \to \infty$ (unendlich dünne Rechtecke) der Näherungswert U_n gegen den exakten Integralwert konvergiert.

Die Anwendung dieses Verfahrens soll an einem Beispiel illustriert werden. Man betrachtet das Intervall $[a, b] = [0, 1]$ und die Funktion $f(x) = \frac{1}{1+x^2}$. Gesucht ist das Integral $\int_0^1 \frac{1}{1+x^2}$. Für diesen Fall ist die exakte analytische Lösung bekannt

$$\int_0^1 \frac{1}{1+x^2} = \arctan(1) - \arctan(0) = \frac{\pi}{4} - 0 \approx 0,785398$$

Für die numerische Berechnung unterteilt man das Intervall $[0, 1]$ in n (z.B. $n = 10$) Teilintervalle und berechnet dann die Untersumme U_n.

Die Numerische Mathematik interessiert sich natürlich auch für den Fehler, den man bei der Diskretisierung notwendigerweise macht. Der *Diskretisierungsfehler* e_n, der vom Diskretisierungsparamter n abhängt, ist definiert als

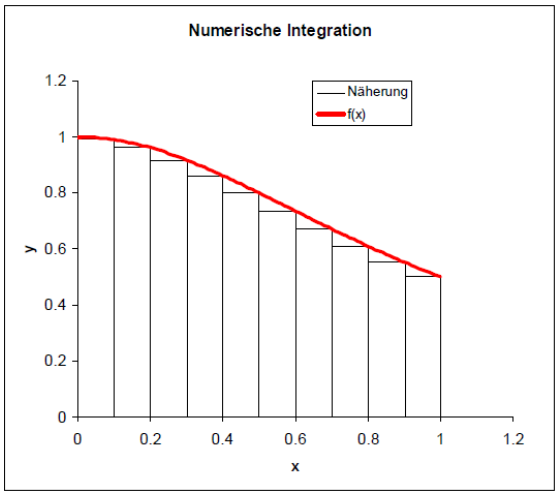


Abbildung 3.4: In diesem Bild sieht man die zehn Rechtecksflächen ($n = 10$) als Annäherung an die Fläche zwischen der Kurve und x-Achse auf dem Intervall $[0, 1]$.

$$e_n = \int_0^1 f(x)\,dx - U_n$$

Anschaulich gibt der Diskretisierungsfehler e_n an, wie viel Fläche zwischen dem Graph der Funktion f und dem Intervall $[0, 1]$ auf der x-Achse nicht von Rechtecken abgedeckt wird. Je kleiner e_n ist, desto besser konnte die numerische Approximation U_n das Integral annähern. Um den Zusammenhang zwischen Diskretisierungsfehler e_n und Diskretisierungsparamter n zu verstehen, ermittelt man für verschiedene Werte von n den Diskretisierungsfehler, der hier in einem bilogarithmischen Maßstab in einem Schaublid präsentiert wird.

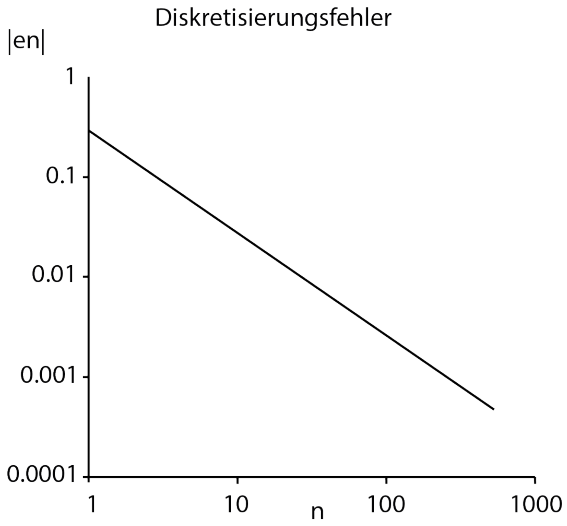


Abbildung 3.5: In diesem Maßstab sieht der Zusammenhang von Diskretisierungsparamter und Diskretisierungsfehler wie eine Gerade aus. Und in der Tat kann man mathematisch beweisen, dass der Fehler e_n für ein ausreichnend großes n immer kleiner wird.

Bei der Fehleranalyse sollte man stets den numerischen Gesamtfehler untersuchen, denn alle Rechnungen auf dem Computer sind neben dem Diskretisierungsfehler auch mit Rundungsfehlern behaftet. Für das obige Beispiel gilt glücklicherweise, dass die Summation in U_n ein numerisch stabiles Verfahren ist, es also wenig empfindlich auf Rundungs- und Eingabefehler ist. Unvermeidbare Rundungs- und Eingabefehler können sich also nicht aufschaukeln. In diesem Fall wird also der Gesamtfehler durch den Diskretisierungsfehler dominiert.

Ziel ist es, mit möglichst wenig Aufwand (abhängig vom Diskretisierungsparameter: Je größer n, desto genauer wird die Integralfläche approximiert, aber auch desto mehr Summanden in U_n muss der Computer berechnen, macht also in Relation gesehen mehr Fehler) einen möglichst geringen Gesamtfehler zu erzielen.

Zur Verdeutlichung wird hier die gleiche Fehlerrechnung wie oben (mit anderen Auswertungspunkten der zu integrierenden Funktion, und zwar jeweils genau in der Mitte jedes Rechteckes) auf einem 32-Bit Rechner durchgeführt, wobei der Diskretisierungsparameter n durch Verdopplung schrittweise von 1 bis 262144 erhöht wird.

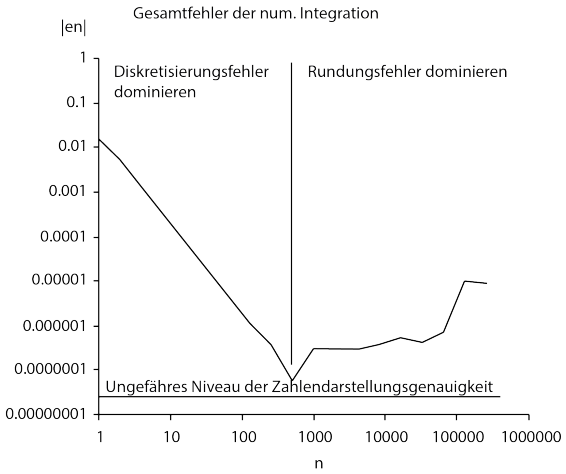


Abbildung 3.6: Diese Abbildung zeigt den unterschiedlichen Einfluss von Diskretisierungs- und Rundungsfehler in Abhängigkeit vom Diskretisierungsparamter n.

Man erkennt deutlich, dass zunächst der Diskretisierungsfehler abnimmt. Ab etwa $n = 512$ gelingt es dem Verfahren jedoch nicht mehr, den Fehler weiter zu verkleinern, denn der Diskretisierungsfehler liegt jetzt bei etwa $1 \cdot 10^{-7}$, was ungefähr in der Größenordnung der unvermeidbaren Eingabefehler liegt. Bei weiterer Erhöhung des Diskretisierungsparameters n schaukeln sich die Rundungsfehler aufgrund der mit n wachsenden Anzahl an ausgeführten Berechnungsschritten langsam auf, der Gesamtfehler wächst also langsam wieder! Allerdings wachsen die Rundungsfehler nur langsam, denn die Summation von Werten gleichen Vorzeichens ist eine numerisch stabile Operation, so dass die numerische Integralberechnung wenig empfindlich auf Rundungs- und Eingabefehler reagiert.
Ziel der Berechnung ist es, mit möglichst geringem Aufwand (gekennzeichnet durch n) einen möglichst geringen Gesamtfehler zu erzielen. Das Experiment zeigt, dass es einen optimalen Diskretisierungsparamter gibt, für den das Verhältnis aus Aufwand und erzielter Ergebnisqualität minimal wird (in Abbildung fünf im Tiefpunkt nach $n = 512$ Schritten). Man kann aus diesem Beispiel schließen, dass in der Numerik „viel“ nicht immer auch gleich „gut“ bedeutet, sondern dass notwendigerweise die Reichweite und Grenzen jedes Verfahrens und seiner Ergebnisse reflektiert werden müssen: Es macht keinen Sinn ein numerisches Verfahren zu verbessern, wenn eine Steigerung des Aufwandes (beschrieben durch einen höheren Diskretisierungsparameter n) keine merkliche Verbesserung des Ergebnisses bewirkt. Im Schaubild oben kann man erkennen, dass ab $n = 1024$ praktisch keine weitere Verbesserung eintritt.

2.4 Methoden

Eine der Hauptaufgaben der Numerischen Mathematik ist es, für die in Anwendungen auftretenden Aufgaben, wie z.B. Integralberechnungen, Ableitungen und Nullstellen von Funktionen, Lösung linearer Gleichungssysteme oder Berechnung von Eigenwerten, stabile

Lösungsalgorithmen zu finden.
Im Folgenden soll exemplarisch der Fokus auf das sogenannte *Explizite Euler-Verfahren* gelegt werden, das die einfachste numerische Lösungsmethode für *Anfangswertprobleme (AWP)* ist.
Ein AWP beschreibt eine Klasse von DGL'en, bei denen der Anfangswert der gesuchten Lösung bekannt ist. Das AWP lautet im Allgemeinen:

$$\begin{aligned} y'(t) &= f(t, y(t)) \\ y(t_0) &= y_0, \end{aligned}$$

wobei y_0 der bekannte Anfangswert der Lösungsfunktion zum Zeitpunkt t_0 ist.

Wie schon in Abschnitt 2.2 erwähnt, besteht die Grundidee fast aller numerischer Verfahren für (gewöhnliche) DGL'en darin, eine Diskretisierung der Zeit auf der x-Achse vorzunehmen und die exakte Lösungsfunktion $y(t)$ dann in endlich vielen Stützstellen mit einer Näherungslösung $\tilde{y}(t) \approx y(t)$ zu approximieren.
Man berechnet also zuerst die Näherungslösung $\tilde{y}(t_1)$ zum Zeitpunkt t_1, als Annäherung für die exakte Lösung $y(t_1)$. Danach benutzt man den gerade zuvor berechneten Näherungswert $\tilde{y}(t_1)$ und ermittelt daraus die nächste Näherungslösung $\tilde{y}(t_2)$, die $y(t_2)$ approximiert. Man fährt so fort für alle weiteren Stützstellen t_i, $i = 1, \cdots, N-1$. usw. für alle Zeitpunkte $t_i, i = 1, \ldots, N-1$.
Würde man alle diese berechneten Funktionswerte in ein Schaubild einzeichnen und die Punkte verbinden, erhielte man eine Annäherung an die gesuchte Lösungsfunktion der DGL (vgl. Abbildung. 3.7)

Wie berechnet man also nun die einzelnen Funktionswerte? Aus der Theorie der DGL'en ist bekannt, dass die exakte Lösung des AWP so ausgedrückt werden kann:

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} f(\tau, y(\tau))\, d\tau$$

Das Integral ist in der Regel nicht exakt (explizit) berechenbar, weil für die Funktion f keine Stammfunktion angegeben werden kann. Deshalb setzt man hier für die numerische Integration eine Approximation ein. Analog zur im Abschnitt 3.2 eingeführten Diskretisierung geht man nun auch hier vor:
Man interpretiert das Integral als Fläche unter dem Graphen der Funktion f und der x-Achse und versucht entsprechend diese Fläche durch eine Reihe von Rechtecken zu approximieren.
Man unterteilt das Zeitintervall $[t_0, T]$ in N Teilintervalle auf und erhält die sogenannte Schrittweite (oder Länge der Teilintervalle und Breite der Rechtecke)

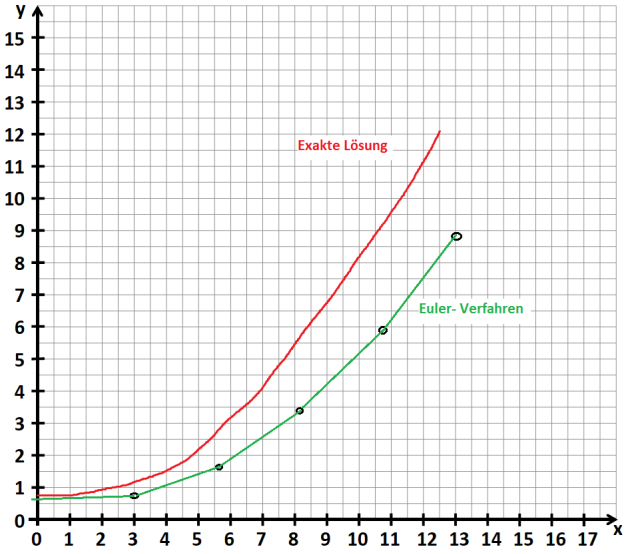


Abbildung 3.7: Die einzelnen berechneten Funktionswerte der Lösungsfunktion werden verbunden und liefern eine Approximation an die exakte Lösung der DGL.

$$h = \frac{T - t_0}{N}$$

und entsprechend N viele diskrete Zeitpunkte:

$$t_i = t_0 + h \cdot i, \quad (i = 1, \ldots, N)$$

Für jede Berechnung eines neuen Funktionswertes $y(t_{i+1})$ muss man das Integral nur auf einem Zeitintervall $[t_i, t_{i+1}]$ lösen, d.h. nur die Fläche eines einzigen Rechteckes berechnen. Die Breite dieses Rechteckes ist immer h, als Höhe wählt man wieder den linken Rand jedes Rechtecks.

Zusammengefasst:

$$\int_{t_i}^{t_{i+1}} f(\tau, y(\tau))\, d\tau \approx h \cdot f(t_i, y(t_i))$$

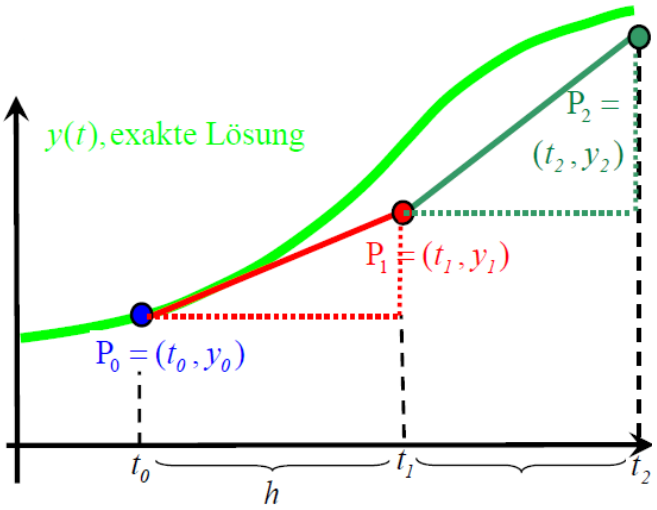


Abbildung 3.8: In der Abbildung werden zwei Euler-Schritte durchgeführt: Mit dem gegebenen Anfangswert (t_0, y_0) kann man den ersten Nährungswert des Eulerverfahrens (t_1 und $y(t_1)$) berechnen, mit diesem dann (t_2 und $y(t_2)$) usw.

Somit hat das erste Rechteck die Höhe $f(t_0, y(t_0))$. Aus der Aufgabenstellung sind bereits t_0 und $y(t_0)$ bekannt. Der erste Nährungswert $\tilde{y}(t_1)$ lautet an der Stelle $t_1 = t_0 + h$

$$\tilde{y}(t_1) = y(t_0) + h \cdot f(t_0, y(t_0))$$

.

Hat man $\tilde{y}(t_1)$ berechnet, kann man in gleicher Art den nächsten Wert $\tilde{y}(t_2)$ berechnen:

$$\begin{aligned} t_2 &= t_1 + h \\ \tilde{y}(t_2) &= \tilde{y}(t_1) + h \cdot f(t_1, \tilde{y}(t_1)) \end{aligned}$$

Damit haben wir die Iterationsvorschrift bzw. den Algorithmus für das Eulerverfahren gefunden mit dem wir zu jedem Zeitschritt t_i die Näherungslösung $\tilde{y}(t_i)$ berechnen können:

$$\tilde{y}(t_0) = y(t_0)$$
$$\tilde{y}(t_{i+1}) = \tilde{y}(t_i) + h \cdot f(t_i, \tilde{y}(t_i)), \quad \text{für } i = 0, 1, \ldots, N-1 \quad (\star)$$

Mathematisch kann man beweisen, dass dieses Verfahren konvergent ist, d.h. dass durch das Verkleinern der Schrittweite $h > 0$ die Differenz $\tilde{y}(t_i) - y(t_i)$ zwischen Näherungswert und Wert der exakten Lösung für beliebiges t_i immer kleiner wird. Das Ergebnis $\tilde{y}(t_{i+1})$ ist also tatsächlich eine Approximation an $y(t_{i+1})$.
Es stellt sich aber leider heraus, dass dieses Verfahren nur sehr langsam konvergiert, d.h. man muss sehr feine Gitter wählen, um eine gute Approximation zu erhalten. Hierfür ist aber deutlich mehr Rechenarbeit nötig. Denn je feiner das Gitter ist, desto mehr Additionen und Multiplikationen in $(\star)$ müssen in der immer feineren Unterteilung des Zeitinvervalls berechnet werden.

Das Euler-Verfahren ist ein iteratives Berechnungsvefahren, das pro Zeitschritt $t_i \mapsto t_{i+1}$ die Näherungslösung im nächsten Gitterpunkt berechnet.
Methodische Weiterentwicklungen dieses Verfahrens bestehen im Grunde aus zwei Ideen:

- Die erste Idee ist es, im Iterationsschritt $(\star)$ zur Bestimmung von $\tilde{y}(t_{i+1})$ nicht nur die zuletzt berechneten Werte $\tilde{y}(t_i)$ und $f(t_i, \tilde{y}(t_i))$ zu verwenden, sondern auch auf jene Werte von $\tilde{y}$ und f zurückzugreifen, die in vorangegangenen Iterationen schon berechnet worden sind. Dies führt zu genaueren und besseren Näherungslösungen, weil mehr Informationen verwendet werden. Auf diese Weise erhält man die Klasse der sogenannten *Mehrschrittverfahren*, die über ein verbessertes Konvergenzverhalten verfügen.

- Die zweite Idee besteht darin, in jedem Iterationssschritt die Funktion $f(t_i, \tilde{y}(t_i))$ auf dem Intervall $[t_i, t_{i+1}]$ an mehreren Zwischenstellen auszuwerten. So erhält man die sogenannten *Runge-Kutta-Verfahren*. Auch hier erreicht man eine schnellere Konvergenz im Vergleich zum Euler-Verfahren.

2.5 Algorithmische Komplexität

Für alle entscheidbaren (algorithmischen) Probleme ist es wichtig zu wissen, wie groß der Rechenaufwand für den Computer sein wird und wie lange der Benutzer auf das berechnete Ergebnis warten muss. Die algorithmische Komplexitätstheorie hat daher die Aufgabe, die Effizienz von Algorithmen in Abhängigkeit von der Größe der Eingabedaten zu beurteilen. Die Laufzeit des Algorithmus (die begrenzte Größe des Speicherplatzes wird hier vernachlässigt) liefert ein wichtiges Maß für die Komplexität von Algorithmen. Für die Laufzeitkomplexität wählt man kein konkretes Zeitmaß, da die tatsächliche Ausführungszeit abhängig ist von der konkreten Programmierung, der Prozessorgeschwindigkeit des jeweiligen Computers, der Programmiersprache etc.

Um die Effizienz von Algorithmen sinnvoll beurteilen zu können, benötigt man daher ein Maß der Zeitkomplexität, das unabhängig vom verwendeten Computer und der Programmiersprache ist.

Man definiert daher die *Laufzeit* $T(n)$ eines Algorithmus als eine Funktion, die der Eingabegröße n die Anzahl an Basisoperationen zuordnet, die der Algorithmus zur Berechnung der Lösung benötigt.
Die Definition der Eingabegröße n hängt von der Problemstellung ab. Bei der Suche eines Elementes in einer Liste ist n die Anzahl der Elemente der Liste, bei einer Multiplikation zweier Matrizen ist n gleich der Dimension der Matrix (Zeile- mal Spaltenlänge). Als Basisoperationen bezeichnet man:

1. Arithmetische Operationen: Addition, Multiplikation, Division, Ab-, Aufrunden;
2. Laden, Speichern, Kopieren von Datensätzen fester Größe, Wertzuweisung von Variablen, Funktionsaufrufe.

Zur Vereinfachung nehmen wir an, dass jede dieser Operationen bei allen Operanden gleich viel Zeit benötigt (im Einheitskostenmaß[4]). Damit sind die Kosten der Basisoperationen nahezu unabhängig von der verwendeten Programmiersprache und einfach ablesbar aus dem Pseudocode oder einem Programmstück.

Beispiele

Bei der Multiplikation quadratischer Matrizen (m Zeilen und m Spalten mit der Dimension $n = m^2$) beträgt die Laufzeit $T(n) = n^3$, da hierfür n^3 Additionen und n^3 Multiplikationen nötig sind (im schlechtesten Fall).
Beim expliziten Euler-Verfahren, das weiter oben in Abschnitt 2.4 vorgestellt worden ist, muss der Computer in jedem Iterationsschritt eine Addition, eine Multiplikation und eine Funktionsauswertung von f berechnen. Bei einer Unterteilung des Zeitinvervalls $[t_0, T]$ in n Teilintervalle ergibt sich also für die Laufzeit $T(n) = n(1 + 1 + T(f))$, wobei $T(f)$ die Kosten der Funktionsauswertung bezeichnet.

Bei kleinen Eingabegrößen (z.B. $n = 9$) ist die Untersuchung des Laufzeitverhaltens eher uninteressant. Interessanter ist es zu untersuchen, wie sich die Laufzeit des Algorithmus bei sehr großen Problemgrößen verhält (z.B. bei der Multiplikation Matrizen mit je 10.000 Zeilen und Spalten), die typischerweise in Anwendungen vorkommt. Die Idee ist nun, die Komplexität der Laufzeit-Funktion mithilfe von Funktionenklassen zu klassifizieren. Funktionen, die im Prinzip „gleich schnell“ wachsen, sollen in dieselbe Funktionenklasse eingeordnet werden. In der Mathematik bzw. Informatik benutzt man hierfür die sogannten *Landau-Symbole* ($\mathcal{O}$-Notation).
Zur Erläuterung dieses Konzepts betrachtet man Funktionen $g, h : \mathbb{N} \to \mathbb{R}^{\geq 0}$ von den natürlichen Zahlen ($\mathbb{N}$) in die positiven reellen Zahlen ($\mathbb{R}^{\geq 0}$). Man definiert dann die *Ordnung von* g, kurz $\mathcal{O}(g)$, als die Menge aller Funktionen h, für die folgendes gilt:

[4]Einheitskostenmaß: Annahme, dass jedes Datenelement, unabhängig von seiner Größe denselben Speicherplatz belegt. Damit ist die Größe der Eingabe bestimmt durch die Anzahl der Datenelemente.

„Langfristig“, d.h. für große Eingabewerte n, verhalten sich die Funktionen h wie die Funktion g. Genauer ausgedrückt kann man jede Funktion h nach oben mit der Funktion g abschätzen, kurz: $h(n) \leq c \cdot g(n)$, für eine beliebige Konstante c.
Eine Funktion f ist also von der Größenordnung $\mathcal{O}(g)$ (kurz: $f \in \mathcal{O}(g)$), wenn sich das Wachstumsverhalten der Funktion f für große Eingabewerte n durch die Funktion g beschreiben lässt. g stellt somit eine obere Schranke für f dar.

Die untere Abbildung, in der die Graphen von $c \cdot g(n)$ und $f(n)$ zu sehen sind, verdeutlicht diesen Sachverhalt. Nach rechts wird der Eingabewert n, nach oben die Laufzeit abgetragen. Man kann gut erkennen, dass die Menge $\mathcal{O}(g(n))$ alle Funktionen $f(n)$ enthält, die langsamer oder gleich schnell wie $g(n)$ wachsen.

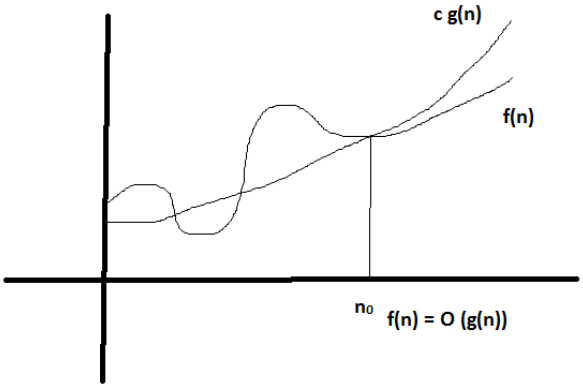


Abbildung 3.9: Für $n > n_0$ an, kann das Verhalten der Funktion $f(n)$ im Wesentlichen durch den Verlauf von $c \cdot g(n)$ beschrieben werden.

Mit der Einführung des Ordnungsbegriffs $\mathcal{O}$ kann man nun Funktionenklassen angeben, um die Komplexität der Laufzeit-Funktion $T(n)$ von Algorithmen zu klassifizieren.

Klasse	Bezeichnung	Wertung	Typische Algorithmen
$\mathcal{O}(1)$	Konstanter Ressourcenverbrauch	Optimal, Selten	Addition von Zahlen, rekursiver Aufruf
$\mathcal{O}(\log n)$	Logarithmisches Wachstum	Sehr günstig	Suchen auf einer sortierten, d.h. geordneten Menge
$\mathcal{O}(n)$	Lineares Wachstum	Günstig	Bearbeiten jedes Elementes einer Menge
$\mathcal{O}(n \log n)$	Leicht überlineares Wachstum	Noch gut	Sortieren einer Liste
$\mathcal{O}(n^2)$	Quadratisches Wachstum	Ungünstig	Primitive Sortierverfahren
$\mathcal{O}(n^3)$	Kubisches Wachstum	Ungünstig	Matrizenmultiplikation
$\mathcal{O}(n^k), k$ fest	Polynomielles Wachstum	Ungünstig	Beispiel: $T(n) = 8n^5 - 2n^3 + n^2 - 3n$
$\mathcal{O}(2^n)$	Exponentielles Wachstum	Katastrophal	Kombinatorische Optimierungsprobleme

Beispiel: Minimum-Suche

In einem Array (hier ein Liste von Zahlen) soll das Minimum, also die kleinste Zahl gefunden werden.

- Eingabe: Array von n Zahlen $(a_0, \ldots, a_{n-1})$
- Ausgabe: Index i, so dass $a_i \leq a_j$ für alle Indizes $1 \leq j \leq n$
- Beispiel: $n = 6$
  - Eingabe: $a_0 = 31, a_1 = 41, a_2 = 59, a_3 = 26, a_4 = 51, a_5 = 48$
  - Ausgabe: $i = 3$

Der Pseudocode des Algorithmus für das Array $(a_0, a_1, a_2, a_3, a_4, a_5)$ sieht folgendermaßen aus:

```
1: procedure
2:     define min(A):
3:     min=0
4:     for  j in range( 1, len(A) ):  do
5:         if A[j] < A[min]:
6:         min = j
7:     return min
```

Algorithmus 1: Suche des Minimums

Nun soll die Laufzeit-Komplexität dieses Algorithmus untersucht werden. Die Laufzeit $T(n)$ ist definiert als Anzahl der Basisoperationen, die der Algorithmus zur Berechnung der Lösung bei einer Eingabegröße n benötigt. Hierfür müssen die „Kosten“ in jeder Zeile festgestellt werden.

Zeile	Kosten	max. Anzahl
3	c_1	1
4	c_2	$n-1$
5	c_3	$n-1$
6	c_4	$n-1$

Für die gesamte Laufzeit $T(n)$, in Abhängigkeit von der Eingabe n = Größe des Arrays, ergibt sich also:

$$T(n) = c_1 + (n-1) \cdot (c_2 + c_3 + c_4) \leq c_1 + n \cdot c_5$$

wobei c_5 irgendeine passende Konstante ist, sodass die Abschätzung oben richtig ist. Man könnte z.B. einfach setzen $c_5 = c_2 + c_3 + c_4 + 1$. Also gilt $T(n) \approx c_1 + n \cdot c_5$.
In welcher Komplexitätsklasse liegt dieser Algorithmus? Die Laufzeit $T(n)$ verhält sich für große Eingabegrößen n im Wesentlichen wie die Funktion $f(n) = c_1 + n \cdot c_5$, die linear in n ist. Damit liegt also die Laufzeit $T \in \mathcal{O}(n)$ und kennzeichnet damit ein lineares Wachstum (vgl. Tabelle).

Man sagt, ein Algorithmus mit Laufzeit-Komplexität $T(n)$ braucht höchstens *polynomiale Rechenzeit*, falls $T(n)$ die Form eines Polynoms $P(n)$ hat, das aus Potenzen von n mit Koeffizienten davor aufgebaut ist:

$$P(n) = a_0 + a_1 n^1 + \ldots + a_{k-1} n^{k-1} + a_k n^k,\ a_i \in \mathbb{N}, k \in \mathbb{N},$$

Wenn $T(n)$ die Form eines Polynoms hat, dann gilt $T(n) \in \mathcal{O}(n^k)$.
Ein Algorithmus braucht höchstens *exponentielle Rechenzeit*, falls es eine Konstante $a \in \mathbb{R}^{\geq 0}$ gibt, sodass $T(n) \in \mathcal{O}(a^n)$.

Es hat sich in der Praxis gezeigt, dass die Grenze zwischen Problemen, die in polynomialer Zeit lösbar sind und denjenigen, die einen exponentiellen Zeitaufwand erfordern, von grundlegender Bedeutung ist. Probleme, die einen polynomialen Aufwand erfordern, sind in einem Zeitrahmen lösbar, der tolerierbar ist. Man nennt deshalb diese Algorithmen praktisch berechenbar (obwohl bereits die Komplexitätsklasse $\mathcal{O}(n^3)$ ungünstig ist), während Algorithmen mit exponentieller Laufzeit als nicht (mehr) praktisch berechenbar gelten.
Anschaulich wird dies zum Beispiel beim sogenannten „Problem des Handlungsreisenden“ (Traveling Salesman Problem, TSP), das zu den kombinatorischen Optimierungsproblemen gehört. Dabei geht es darum, bei einer vorgegebenen Anzahl an Städten diejenige Rundreise (die Stadt zu Beginn der Reise ist zugleich die letzte Station der Reise) auszuwählen, sodass die gesamte Reisestrecke des Handlungsreisenden möglichst kurz ist. Jede Stadt soll dabei genau einmal besucht werden. Es existiert kein Algorithmus, der eine kürzeste Rundreise in polynomialer Laufzeit bestimmen kann. Bei der Auswahl von n Städten hat der Handlungsreisende in jedem Schritt diejenigen Städte als nächsten Zielort zur Auswahl, die er noch nicht besucht hat. Bei n Städten sind das $(n-1)! = 1 \cdot 2 \cdot \ldots \cdot (n-1)$ unterschiedliche infrage kommende Touren: Jede Rundreise muss in einer Stadt beginnen, die restlichen $(n-1)$ Städte müssen in irgendeiner Reihenfolge besucht werden, bevor man wieder zur ersten Stadt zurückkehrt. Es gibt aber genau $(n-1)!$ viele Möglichkeiten, die restlichen $(n-1)$ Städte nacheinander aufzusuchen.

Städte	mögliche Rundreisen	Laufzeit
3	1	1 msec
4	3	3 msec
5	12	6 msec
6	60	60 msec
7	360	360 msec
8	2.520	2,5 sec
9	20.160	20 sec
10	181.440	3 min
11	1.814.400	0,5 Stunden
12	19.958.400	5,5 Stunden
13	239.500.800	2,8 Tage
14	3.113.510.400	36 Tage
15	43.589.145.600	1,3 Jahre
16	653.837.184.000	20 Jahre

Die obige Tabelle zeigt, dass der Raum der Möglichkeiten überexponentiell mit der Anzahl der Städte wächst. Die letzte Spalte gibt eine Abschätzung für die Laufzeit eines Algorithmus an, der das TSP mithilfe der Holzhammermethode (jede mögliche Rundreise muss hinsichtlich ihrer Länge tatsächlich im Algorithmus berücksichtigt werden) zu lösen versucht. Hierbei wird angenommen, dass die Bearbeitung einer einzigen Rundreise etwa eine Millisekunde Zeit benötigt. Aus der letzten Zeile kann man ablesen, dass ein solcher Algorithmus für die Lösung des TSP mit nur 16 Städten mehr als eine halbe Billion

Rundreisen betrachten muss und so mehr als 20 Jahre Rechenzeit benötigt.
Selbst wenn man klügere Algorithmen mit kürzerer Laufzeit zur Hand nimmt (die nicht „blind“ alle Möglichkeiten durchprobieren), schaffen sie es dennoch nicht, die Laufzeit langsamer als exponentiell mit der Anzahl der Städte anwachsen zu lassen.

2.6 Grenzen der numerischen Approximation

Seit dem Aufkommen der immer leistungsfähigeren elektronischen Rechenanlagen wird die numerische Mathematik immer wichtiger. Seit Mitte der 1980er Jahre wurden bis heute wirkmächtige Methoden entwickelt, mit denen Lösungen mathematischer Problemstellungen effektiv durch auf Digitalrechnern realisierbaren numerischen Algorithmen angenähert werden können. So kann man mittlerweile teure Experimente, wie z.B. Windkanalversuche bei der Flugzeugkonstruktion, durch beliebig oft und schnell wiederholbare Simulationsrechnungen ersetzen.

Von zentraler Bedeutung bei der numerischen Approximation ist die Bewertung der Güte der berechneten Ergebnisse. Wie sehr kann man den Ergebnissen, die ein Algorithmus auf einer Maschine ausgerechnet hat, „vertrauen“? Denn Fehlerquellen lauern in der Wahl des mathematischen Modells, in der Formulierung des Algorithmus, in der Implementierung auf dem Computer und bei der Interpretation der Resultate.
Die Zuverlässigkeit kann mit zwei Methoden untersucht werden, die Verifizierung und Validierung genannt werden.
Die *Verifizierung* beschäftigt sich mit der Frage, ob das ausgewählte mathematisch-physikalische Modell korrekt gelöst wird („solving the equations right“). Man versucht zu klären, inwiefern die Implementierung des Computercodes auf dem Computer das zugrunde gelegte mathematisch-physikalische Modell repräsentiert („code implementation“) und wie nahe das numerische Ergebnis der exakten Lösung der DGL'en im Modell kommt („solution implementation“).

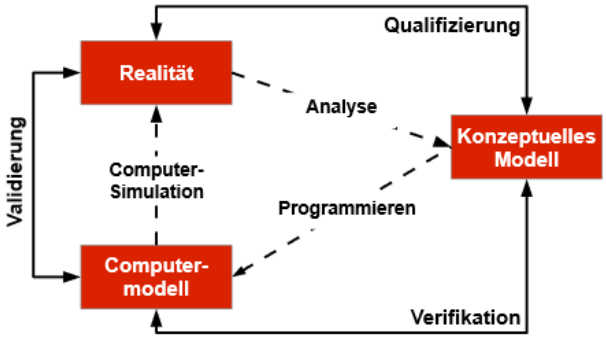


Abbildung 3.10: Verifikation und Validierung in der Modellierung und Simulation nach Oberkampf und Roy.

Aufgaben der Verifikation sind also die Untersuchung der numerischen Algorithmen und des Computercodes. Hierzu müssen die folgenden numerischen Fehler berücksichtigt werden:

- Diskretisierungsfehler: Kontinuierliche Prozesse werden durch endliche ersetzt (z.B. Approximation des Integrals durch endlich viele Summen).
- Abbruchfehler: Unendliche in so genannten Wiederholungsschleifen laufende Algorithmen müssen nach endlich vielen Schritten abgebrochen werden.
- Rundungsfehler: Auf dem Computer müssen alle Rechnungen auf einem endlichen Zahlbereich durchgeführt werden (z.B. $1/3 \approx 0,33$). Jede Zahl, die in den Computer eingegeben und durch Rechnungen verändert wird, muss gerundet werden.

Oft ist der Vergleich mit der exakten Lösung des Modells hinfällig, da diese oft überhaupt nicht bekannt ist bzw. wir nicht im Stande sind, sie mathematisch zu berechnen, worin ja letztlich der Grund bestand, überhaupt den Computer rechnen zu lassen. Ein gangbarer Weg besteht deswegen darin, zu untersuchen, ob und wie schnell die numerischen Lösungen konvergieren, wenn man das Gitter, auf dem der Computer rechnet, immer feiner macht (eine Hinzunahme von immer mehr Stützstellen, an denen man Rechnungen durchführt). Fragen der Verifikation zielen also grundsätzlich auf das Verhältnis zwischen mathematisch-physikalischem Modell und Computermodell ab.
Ziel der *Validierung* hingegen ist es zu fragen, ob denn die „richtigen“ Gleichungen gelöst werden bzw. ob das „richtige“ Modell verwendet wird („solving the right equations“), um das Phänomen in der Welt adäquat zu beschreiben, das man mit der Wahl eines Modells besser verstehen möchte. Taktisches Ziel ist somit, Fehler und Ungenauigkeiten im Computermodell wie auch in den experimentellen Daten (wenn diese zur Verfügung stehen) herauszustellen und sie zu verkleinern. Damit wird das Vertrauen in die Vorhersage-Fähigkeit des Computermodells erhöht. Die Validierung erhöht also das Vertrauen in die modellabhängigen und simulationsgenerierten Daten, in dem Sinne, dass sie nicht zu stark von jenen Verhältnissen des realweltlichen Systems abweichen.
Erster Ansatz ist hier das Durchführen sogenannter *Validierungsexperimente*, um beurteilen zu können, wie gut ein physikalisches Modell einen Wirklichkeitsausschnitt (die zu untersuchenden Aspekte des realweltlichen Zielsystems) repräsentieren kann. Hierbei handelt es sich um Simulationsexperimente, mit denen mittels Simulationstechnik das Verhalten von Comutermodellen erforscht wird (sie werden auch als Experimente mit Modellen, als Experimentieren im Virtuellen ohne Materialität aufgefasst).
Validationsexperimente sollten gleichermaßen von Modellierern wie von Experten des Experimentalsystems entwickelt werden, um sicherzustellen, dass das verwendete Computermodell hinreichend genaue Aussagen über das Realsystem erlaubt. Ebenso sollten Validierungsexperimente so konzipiert werden, dass alle relevanten physikalischen Bedingungen wiedergegeben werden.

In der Strömungsmechanik (z.B. bei Flugzeugen) bilden verkleinerte Prototypen im Windkanal ein Experimentalsystem. Das zugehörige Computermodell fußt mathematisch auf den Navier-Stokes-Gleichungen, die in Abschnitt 1.3 eingeführt worden sind.

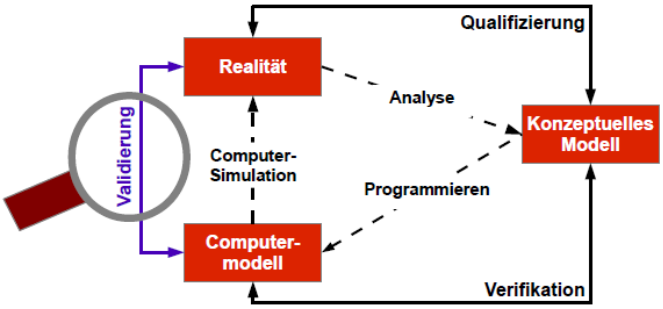


Abbildung 3.11: Validierung im Überblick nach Oberkampf und Roy.

Ziel der sogenannten *Kalibrierung* ist es nun, die Messdaten des Experimentalsystems x_{exp} mit den simulationsgenerierten Ergebnissen x_{sim} zu vergleichen bzw. die Differenz von x_{exp} und x_{sim} zu reduzieren. Falls der Unterschied als zu groß bewertet wird, nimmt man mit der Kalibrierung gezielte Änderungen am Modell vor, um es dem Realsystem anzupassen. Für die Messung dieses Unterschieds benötigt man zuerst einmal ein Maß, eine sogenannte Validationsmetrik d, die einen quantitativen Vergleich von x_{exp} und x_{sim} erlaubt. Man spricht dann von Realitätstreue oder Gültigkeit/Validität, wenn die Differenz in der gewählten Metrik $d(x_{\text{exp}}, x_{\text{sim}})$ klein genug ist. Was "klein genug" bedeutet, hängt stets vom betrachteten Modell ab und davon mit wie viel Restunsicherheit man sich zufrieden geben möchte. Validität eines Modells ist also nie absolut definierbar, vielmehr ist Validierung immer Modell-spezifisch zu betrachten: Je nach Anforderung und Zielsetzung der Modellierung wird Validität unterschiedlich festgelegt, d.h. wie viel Abweichung $d(x_{\text{exp}}, x_{\text{sim}})$ man noch toleriert, um von Realitätstreue zu sprechen.
Sind genug Messdaten des Experimentalsystems vorhanden (sodass man sinnvoller Weise einen Vergleich mit simulationsgenerierten Ergebnissen anstellen kann), können immer noch eine Reihe von Problemen auftreten. Die gemessenen Daten des Experimentalsystems können zum Beispiel gestört oder verrauscht sein. Falls keinerlei Messdaten des zu modellierenden Experimentalsystems vorliegen, müssen für die Validierung Daten aus ähnlichen Systemen, Schätzungen oder sogar Daten aus anderen Modellen verwendet werden. Diese Daten sind oftmals ungenauer und damit weniger repräsentativ für das zu modellierende System.

Schritte zur Validierung von Modellen

Ziel der Kalibrierung ist die Identifikation und Beseitigung/Minimierung von Verhaltensunterschieden zwischen realem System (kurz System) und Modell M. Hierfür muss das Modell angepasst, d.h. geändert werden. Es gibt zwei Klassen möglicher Modellanpassungen:

- *Struktur-Änderungen* modifizieren die Struktur des (Simulations-)Modells, indem der Programm-Code verändert wird und neue Programmzeilen hinzugenommen oder vorhandene Abläufe modifiziert werden.

- *Parameter-Änderungen* modifizieren Werte der einzelnen Modellparameter. Dazu sind keine Änderungen an der Struktur des Simulationsprogramms notwendig.

Parameter-Änderungen sind in den meisten Fällen einfacher auszuführen als Struktur-Änderungen. Bei der Modellierung komplexer Systeme erfordert die Kalibrierung aber in fast allen Fällen auch Strukturveränderungen. Bevor Änderungen am Modell vorgenommen werden können, müssen erst die (möglichen) Ursachen für die Verhaltensunterschiede, gemessen mit $d(x_{\text{exp}}, x_{\text{sim}})$, abgeleitet werden. In beiden Fällen ist es ratsam, den Bereich, in dem Probleme wahrscheinlich auftreten, möglichst frühzeitig einzugrenzen.
Endlich viele (hier k-viele) Anpassungen der Modellstruktur kann man so auffassen, dass hierdurch neue Modelle $M_1, \ldots, M_k$ mit den zugehörigen Ergebnissen aus den Simulationsexperimenten $x_{\text{sim},1}, \ldots, x_{\text{sim},k}$ entstehen. Über den Vergleich in der Metrik $d(x_{\text{exp}}, x_{\text{sim},i})$, $(i = 1, \ldots, k)$ kann man dann entscheiden, ob ein Modell besser als ein anderes ist.
Bei Parameter-Änderungen steht die Anpassung von Parametern im Mittelpunkt. Wenn $M(p)$ das Modell mit Parametervektor p ist (und die computermodelgenerierten Daten mit $x_{\text{sim},p}$ bezeichnet werden), so wird ein neuer Parametervektor p^* gesucht, sodass $d(x_{\text{exp}}, x_{\text{sim},p^*})$ klein wird.

Nun sei einmal angenommen das Ziel wäre erreicht und ein Modell M für ein System gefunden, sodass $d(x_{\text{exp}}, x_{\text{sim}}) < \epsilon$ für eine akzeptable Fehlerschranke ϵ gelte. Ist damit bereits eine Deckungsgleichheit von Modell und Experiment erreicht, sodass Folgerungen aus Experimenten identisch sind zu Folgerungen aus Modell-Experimenten? Dies kann aus den folgenden Gründen bezweifelt werden: Über Änderungen am Modell wurde das Modell- und das Simulationsverhalten, jeweils für einen Zustand der Umwelt bzw. einen Zustand des Experimentalsystems angepasst. Anspruch und Ziel von Simulationsexperimenten ist die Generierung von Aussagen über andere Situationen, für die das Experimentalsystemverhalten (noch) unbekannt ist. Das hier auftretende Problem wird in Abbildung 3.12 verdeutlicht.

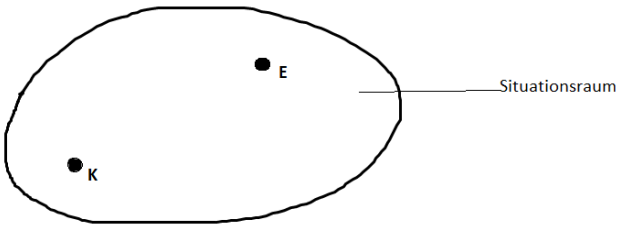


Abbildung 3.12: Situation nach der Validierung

Die Menge aller Situationen (Situationenraum) wird in der Abbildung zweidimensional dargestellt. Für einen Punkt K in diesem Situationenraum wurde die Nähe des Modell-

verhaltens zum Verhalten im Experiment nachgewiesen. Als Ersatz für das System soll das Modell M nun aber an einem Punkt E im Situationenraum angewendet werden. Ist nun der Unterschied zwischen Modell- und Systemverhalten an der Stelle E vergleichbar gering wie bei K? Im Allgemeinen für einen beliebigen Punkt E im Situationenraum ist dies sicher nicht richtig. Wahrscheinlich ist, dass es einen Gültigkeitsbereich G gibt, indem $d(x_{\text{exp}}, x_{\text{sim}}) < \epsilon$ und einen Bereich H, in dem $d(x_{\text{exp}}, x_{\text{sim}}) \geq \epsilon$ gilt.
Zu entscheiden ist also, ob E innerhalb oder außerhalb von G liegt. Diese Frage kann nur beantwortet werden, wenn vom Verhalten in einer Situation K auf das Verhalten in anderen Situationen, wie E, geschlossen werden kann. Aber wie soll dies gelingen? Denn im positiven Fall würde dies bedeuten, dass Aussagen über Systemverhalten in der Welt ohne den Abgleich mit experimentellen Daten aus dem Realsystem abzuleiten wären. Bei nahezu allen realen Systemen ist dies aber nicht möglich. Die Grenzen eines Gültigkeitsbereiches können also nicht bewiesen werden, sondern es kann nur ein Vertrauen in Umfang und Grenze des Gültigkeitsbereiches erreicht werden. Dies bedeutet eine Abschwächung des Begriffes der Validität für ein Modell. Dennoch können aus der Gültigkeit für eine Situation K nützliche Informationen gewonnen werden. Es ist anzunehmen, dass Situationen nahe K eher zu G gehören, als Situationen die sich deutlich von K unterscheiden. Dies ist in Abbildung 3.13 dargestellt, wobei es nicht so sein muss, dass der Gültigkeitsbereich stets einen zusammenhängenden Bereich darstellt, wie in der Abbildung gezeichnet.

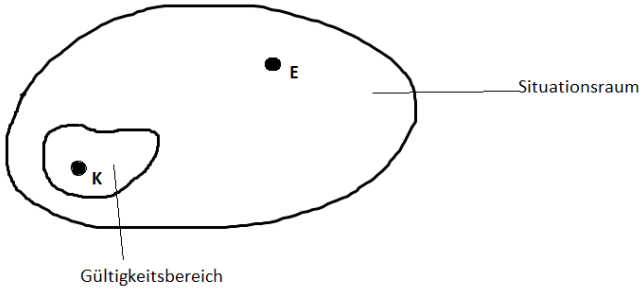


Abbildung 3.13: Validierung mit Gültigkeitsbereich

Bis hierhin wurde ein Modell M gefunden, das nur solche Verhaltensabweichungen liefert, die unterhalb einer tolerierbaren Schwelle liegen ($d(x_{\text{exp}}, x_{\text{sim}}) < \epsilon$. Der nächste Schritt könnte nun darin bestehen, diesen Vorgang zu wiederholen, um durch zusätzliche Modifikationen M so zu verbessern, dass $d(x_{\text{exp}}, x_{\text{sim}})$ weiter reduziert wird, um zu einem noch besser angepassten Modell zu gelangen. In der Praxis zeigt sich aber, dass dieses Vorgehen oft nicht zu einer Vergrößerung des Gültigkeitsbereiches G führt. Im Gegenteil beobachtet man bei einem solchen Vorgehen, das Problem der Überanpassung (overfitting). Dabei werden zwar für die momentane Kalibriersituation (für einen vorgegebenen Datensatz) die Verhaltensunterschiede reduziert, sie steigen dafür aber gleichzeitig für andere Situationen (andere Datensätze) an. Dies führt dazu, dass gerade bei sehr komplexen Modellen mit

vielen Parametern sich im Prinzip jedes Verhalten anpassen lässt, dies aber nichts über die Validität des Modells auf einem größeren Bereich aussagt.

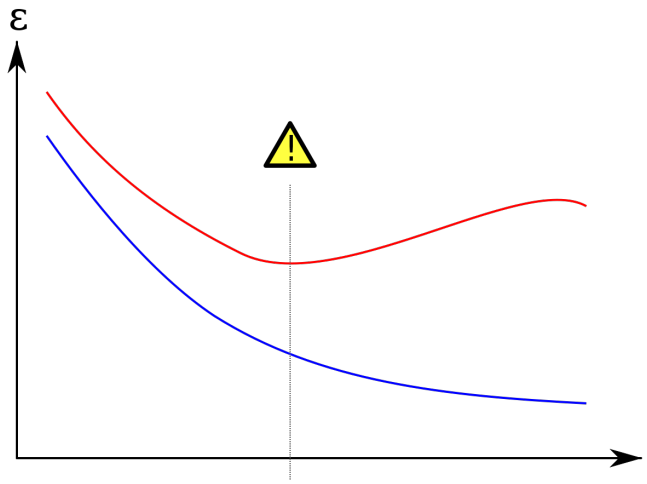


Abbildung 3.14: blau: Fehler bzgl. Trainingsdatensätzen in der Kalibriersituation; rot: Fehler bzgl. Testdatensätzen anderer Situationen. Wenn der Fehler bzgl. der Testdatensätze steigt, während der Fehler bzgl. der Trainingsdatensätze fällt, dann befindet man sich möglicherweise in einer Überanpassungssituation.

Der Gefahr der Überanpassung kann man dadurch begegnen, dass man nicht nur an einer Stelle, sondern an mehreren Stellen kalibriert (dies setzt voraus, dass entsprechende Trainingsdatensätze vorhanden sind). Es bleibt dennoch zu zeigen, dass keine Überanpassung erfolgt ist.
Das Vertrauen in das Modell kann nur erhöht werden, wenn die Gültigkeit für Situationen nachgewiesen wird, die nicht bereits zur Kalibrierung verwendet wurden. Diese Überprüfung ist die eigentliche Validierung. Man prüft hier, ob die Verhaltensunterschiede zwischen Modell und Realsystem für Validiersituationen ähnlich sind, wie für Kalibriersituationen. Im Gegensatz zur Kalibriersituation führt das Ergebnis einer Validiersituation nicht automatisch dazu, dass das Modell an den vorliegenden Datensatz angepasst wird. Ausschließlich eine Kalibrierung ohne Validierung durchzuführen ist daher in vielen Fällen nicht ratsam. Die folgende Abbildung 3.15 verdeutlicht das Vorgehen.

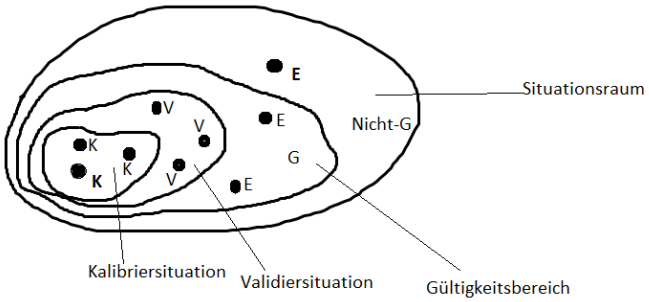


Abbildung 3.15: Kombinierte Kalibrierung und Validierung.

Es bleibt dennoch ungewiss, ob eine unbekannte Experimentsituation E innerhalb oder außerhalb des Gültigkeitsbereichs G liegt. Dies kann man nur rückwirkend nachweisen, was aber für Vorhersagen nicht interessant ist. Trotzdem erhöht die Validierung, auch wenn sie retrospektiv ist, das Vertrauen in ein Modell. Unklar bleibt hingegen, ob Experimentalsituationen E, die weit weg von Kalibrier- und Validiersituation liegen, mit dem Modell hinreichend beschrieben werden können. Meist sind dies aber genau jene Situationen, für die man auf die Prognose-Fähigkeit des Modells zählt.

Literatur

Deuflhard, Peter und Folkmar Bornemann (2008). *Gewöhnliche Differentialgleichungen: Ordinary differential equations*. 3., durchges. und korrigierte Aufl., [elektronische Ressource]. Bd. / Peter Deuflhard ... ; 2. de Gruyter Lehrbuch. Berlin: de Gruyter. ISBN: 3110203561. DOI: 10.1515/9783110203578. URL: http://www.reference-global.com/doi/book/10.1515/9783110203578.

Oberkampf, William L. und Christopher J. Roy (2012). *Verification and validation in scientific computing*. Cambridge: Cambridge Univ. Press. ISBN: 0521113601.

Rolf Rannacher (2005). Heidelberg. URL: http://numerik.iwr.uni-heidelberg.de/~lehre/notes/num0/numerik0.pdf.

Schöning, Uwe (2009). *Theoretische Informatik - kurz gefasst*. 5. Aufl., [Nachdr.] HochschulTaschenbuch. Heidelberg: Spektrum Akad. Verl. ISBN: 3827418240. URL: http://deposit.d-nb.de/cgi-bin/dokserv?id=3030761&prov=M&dok_var=1&dok_ext=htm.

3 Implementierung

Nico Formanek

3.1 Was ist die Aufgabe der Implementierung?

Da die Praktiker Implementierung in zwei verschiedenen Weisen gebrauchen, ist eine Unterscheidung angebracht. Zum einen meint Implementierung den Prozess von einer Idee zur Software, zum Anderen das Verhältnis zwischen einem abstrakten Objekt, z.B. einem Algorithmus und dessen Konkretisierung in einer Programmiersprache. In diesem Abschnitt beschäftigen wir uns mit Implementierung in der ersten Verwendungsweise.
Eine Idee macht noch keine funktionierende Software. Der im Kapitel über Monte-Carlo Simulationen angeführte Pseudo-Code lässt sich zwar von Menschen verstehen, aber noch nicht auf Computern ausführen. Hierfür muss der Pseudo-Code erst in eine Programmiersprache übertragen, mittels eines Compilers in Machinencode übersetzt und schlussendlich vom Betriebssystem ausgeführt werden. Jeder dieser Übergänge stellt eine Fehlerquelle dar. Die häufigsten Fehler treten beim Übertragen der Idee in eine Programmiersprache auf, weshalb Programmierer häufig mehr Zeit mit dem Überprüfen (debuggen) des Codes verbringen, als mit der eigentlichen Programmierung. Unter dem Begriff *software engineering* oder *Softwaretechnik* firmieren verschiedene Methoden, die bei komplexen Programmierprojekten helfen sollen Fehler zu verringern.
Vieles am Implementierungsprozess lässt sich als Optimierung verstehen. Die Auswahl der Programmiersprache, kann nach Kriterien der Performanz, Plattformunabhängigkeit, Verfügbarkeit von Bibliotheken, sowie der Verfügbarkeit von Programmieren für spezielle Programmiersprachen geschehen. Für die Auswahl von Compiler und Betriebssystem gilt ähnliches, wobei hier meistens nicht so viele Auswahlmöglichkeiten bestehen. Man erkennt leicht, dass es sich um ein Problem der multiplen Realisierbarkeit handelt. Alle Programmiersprachen können rein formal die gleichen Berechnungen durchführen, sie sind Turing-vollständig. Hinzu kommt, dass die Lösung einer Programmieraufgabe verschiedene Algorithmen zulässt. Zudem lässt sich ein Algorithmus in einer Programmiersprache verschiedentlich darstellen.
Betrachten wir als Beispiel die Sortierung eines Arrays von natürlichen Zahlen. Wir entscheiden uns für den simplesten Sortieralgorithmus namens Bubble Sort. Bubble Sort beginnt am Anfang des Arrays und vergleicht benachbarte Zahlen. Falls sie in der falschen Reihenfolge sind, werden sie vertauscht. Der Pseudocode lautet wie folgt:

```
function BUBBLESORT(A)
    n ← length(n)
    repeat
        swapped ← false
        for i = 1 ∧ i ≤ n − 1 do
            if A[i − 1] > A[i] then
                swap(A[i − 1], A[i])
```

$swapped \leftarrow true$
$i \leftarrow i + 1$
until $swapped = false$

In der Programmiersprache C lässt sich Bubble Sort wie folgt implementieren:

```
void bubblesort(int *array, int length)
{
 int n = sizeof(array);
 int i;
 int tmp;
 int swapped;
   do {
       swapped = 0;
       for(i=0; i < n-1; i++)
        {
           if(array[i-1] > array[i])
            {
               tmp = array[i];
               array[i] = array[i-1];
               array[i-1] = tmp;
               swapped = 1;
            }
        }
   } while(swapped == 0);
}
```

Eine weitere Möglichkeit den Algorithmus in der Programmiersprache C zu implementieren, die ohne die while-Schleife auskommt:

```
void bubblesort(int *array, int length)
{
    int i, j;
    for (i = 0; i < length - 1; ++i)
    {

        for (j = 0; j < length - i - 1; ++j)
        {
            if (array[j] > array[j + 1])
            {
                int tmp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = tmp;
            }
```

```
            }
        }
    }
```

Je nachdem welche Optimierungen der Compiler vornimmt, kann der Machinencode dieser Programme sogar identisch sein. Hierauf hat ein Programmierer wenig Einfluss, der Maschinencode wird, außer in Ausnahmefällen, von ihm nicht angesehen. Wie leicht vorstellbar, sind die Möglichkeiten der Realisierung bei komplexeren Programmen immens. Aufgabe der Implementierung ist es die Beste auszuwählen und dabei möglichst wenige Programmierfehler zu machen.

3.2 Randbedingungen für HPC

Im Bereich des Hochleistungsrechnen (HPC) gelten andere Bedingungen als in der Softwareentwicklung für Endverbraucher. Ein Grund für die Einführung von Programmiersprachen war, neben der leichteren Lesbarkeit, die Möglichkeit den gleichen Code für unterschiedliche Betriebssysteme und unterschiedliche Hardware zu benutzen. Der C-Code von Bubble Sort aus dem obigen Beispiel lässt sich ohne Veränderung auf Linux und Windows kompilieren und ausführen. Einzig der Compiler ist spezifisch für die jeweilige Hardware und das Betriebssystem. Im Massenmarkt von Software ist es ein unschätzbarer Vorteil, wenn z.B. eine App sich ohne großen Arbeitsaufwand für Android und iOS veröffentlichen lässt. Teilweise geht man sogar dazu über komplett plattformunabhängigen Code zu produzieren.
Es gibt verschiedene Gründe warum dies im HPC nicht möglich ist. Höchstleistungsrechner sind meistens auf ein bestimmtes Anwendungsproblem optimiert und damit Insellösungen. Ihre Hardware ist z.B. für die Lösung von strömungsmechanischen Problemen optimiert, was Performanceeinbußen bei der Simulation von Klimaphänomenen zur Folge haben kann. Die Abstraktionsebenen Betriebssystem und Compiler sind durchlässiger, d.h. Programmierer müssen die genaue Konfiguration der Hardware kennen und für diese programmieren, um optimale Effizienz zu erreichen. Aktuelle Systeme haben ca. 20000 CPUs, die parallel programmiert werden müssen, auch die Speicherlösungen unterscheiden sich von Rechner zu Rechner. Diese Hardwareunterschiede ziehen Unterschiede im Betriebssystem nach sich, welche wieder Unterschiede in den Compilern nötig machen mit dem Resultat, dass Software nur noch auf dem passenden Hochleistungsrechner läuft. Natürlich gibt es Programme die auf allen Hochleistungsrechnern benutzt werden – will man z.B. in die TOP500 Liste kommen, muss man das Linpack Benchmark ausführen können. Dies ist ein Benchmark, was lineare Gleichungssysteme löst und dabei misst wieviele Gleitkommaoperationen der Rechner pro Sekunde ausführen kann. Linpack muss für jeden Hochleistungsrechner neu angepasst und kompiliert werden. Durch die hardwarenähere Programmierung im HPC Bereich konnten sich Paradigmen wie Objektorientierung nicht durchsetzen, da die durch sie eingeführten Abstraktionen Performanzeinbußen zur Folge haben. Außerdem werden weiterhin ältere Programmiersprachen wie Fortran verwendet.

Die Notwendigkeit für viele Prozessoren zu programmieren, schafft das Problem der Parallelisierung. Viele Programmieraufgaben sind ihrer Natur nach seriell und nicht trivial parallelisierbar. Eine Ausnahme bilden unter anderem Berechnungen mit Matrizen und Vektoren. Als Beispiel für ein trivial parallelisierbares Problem sei die Vektoraddition genannt. In der Programmiersprache C, könnte man die nicht-parallele Addition zweier Vektoren wie folgt implementieren:

```
int* vecAdd(int *a, int *b, int length)
{
    int i.
    int c[length];
    for (i = 0; i < length - 1; i++)
    {

        c[i] = a[i] + b[i]
    }
  return c*;
}
```

In der Programmiersprache CUDA-C, eine Erweiterung von C für parallele Berechnungen auf Graphikkarten, kann man zwei Vektoren wie folgt addieren:

```
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    int id = blockIdx.x*blockDim.x+threadIdx.x;
        c[id] = a[id] + b[id];
}
```

Die Variable *id* steht hier für einen Thread, der auf einem Kern der Grafikkarte läuft und eine Zeile der Addition übernimmt. Durch die parallele Ausführung spart man sich im Vergleich zum gewöhnlichen C Programm die for-Schleife, dadurch wird die Berechnung in einem Schnitt ausgeführt. Allerdings braucht man für die Addition der jeweiligen Einträge im Vektor einen Kern. Man ist also durch die maximale Anzahl dieser beschränkt in der Größe der parallel zu verarbeitenden Vektoren. Dies ist eine der Hardwarelimitationen die Programmierer bei der Implementierung für HPC beachten müssen.

3.3 Der Software-Lebenszyklus in Hochleistungsrechnern

Wie im Wissenschaftsbereich üblich haben Programme für Hochleistungsrechner häufig eine lange Versionsgeschichte. Nutzungsdauern von über 30 Jahren sind keine Seltenheit, die Finite-Elemente-Codes aus dem Apollo Programm der 60er Jahre „leben“ in modifizierter Form heute noch weiter. Typischerweise werden Programme in Teams von fünf bis 20 Personen erstellt, wobei letzteres eher eine Ausnahme im akademischen Bereich

darstellt. Hinzu kommt, dass die Programmierer gleichzeitig die Benutzer ihrer eigenen Programme sind, sodass die Entwicklung von numerischen Algorithmen parallel zur Programmentwicklung stattfindet. Zusätzlich zu den geänderten Programmiererfordernissen von Hochleistungsrechnern wie Parallelismus und Maschinennähe ist das Debugging von Programmen komplizierter. Programme werden auf Hochleistungsrechnern meistens im sogenannten Batch-Queueing ausgeführt, das heißt eine Liste wird mit auszuführenden Programmen gefüllt und je nach Freikapazität des Rechners abgearbeitet. Debuggingergebnisse sind also, anders als im gewöhnlichen Prozess, nicht sofort verfügbar. Ein Debugging des kompletten Programms wird zumeist, wegen der teueren Rechenzeit, durch ein inkrementelles Debugging von Komponenten und Teilroutinen ersetzt. Durch die langen Lebensdauer der Software verfolgen die Programmierer einen Konservatismus bezüglich Programmiersprachen und Algorithmen. Auch wird großer Wert auf die Lesbarkeit des Codes gelegt, da dieser über mehrere Wissenschaftlergenerationen gepflegt werden muss.

4 Darstellungsfragen/ Visualisierung

Die Funktion von Bildern für die Produktion und Organisation von Wissen nimmt insbesondere in der Medizin und Naturwissenschaft eine wichtige Rolle ein. Inbesondere die technologische Entwicklung der letzten drei Jahrzehnten ermöglichte eine dauerhafte Speicherung riesiger Datenmengen durch den Computer. Die wissenschaftliche Visualisierung war über einen langen Zeitraum im Wesentlichen auf zweidimensionale Darstellungen begrenzt.
Erst die Verwendung des Computers bot zu Beginn der 60er Jahre vollkommen neue graphische Darstellungsmöglichkeiten und markierte den Beginn der Computergraphik. Nach kurzer Zeit war die dreidimensionale Darstellung von Daten aus den Naturwissenschaften nicht mehr wegzudenken und es begann sich in diesem Zuge das eigenständige Gebiet der wissenschaftlichen Visualisierung zu bilden. In Abgrenzung zur Informationsvisualisierung (z.B. eine Übersichtskarte der Londoner U-Bahn) wird im Folgenden die *wissenschaftlicher Visualisierung* thematisiert werden, die sich mit strukturierten Daten in Raum und Zeit befasst.

Ausgangsgröße für eine Visualisierung ist zum Beispiel das Simulationsergebniss der diskretisierten Navier-Stokes-Gleichungen für ein bestimmtes Fluid in irgendeiner Umgebung. Diese stellt in der Regel eine sehr große Datenmenge dar. Zur Erinnerung: Die Simulationsergebnisse bestehen im Fall der Navier-Stokes-Gleichungen aus Angaben zur Geschwindigkeitsverteilung $v(\vec{x}, t)$ an einzelnen, endlich vielen Orts- und Zeitpunkten $\vec{x}$ und t (in einem Gitter). Man kann sich das Simulationsergebnis als eine lange Liste von Zahlenwerten vorstellen. Jeder Listeneintrag zeigt die Geschwindigkeit an einem Gitterpunkt (Ort- und Zeitachse) an.
Diese riesige Liste an Daten lässt sich aber oft nicht unmittelbar für die Erkenntnisgewinnung nutzen, weil diese Form der Repräsentation von Daten oft kein tieferes Verständnis über Struktur und Zusammenhänge erlaubt. Ziel der Visualisierung ist es also, wesentliche Eigenschaften von Daten zu erfassen und leicht verständlich darzubieten. Welche Eigenschaften der berechneten Daten als „wesentlich“ betrachtet werden, ergibt sich aus der jeweiligen Anwendung. In der Strömungsmechanik möchte man zum Beispiel die Charakteristika komplexer, oft instationärer Strömungen herausfiltern und sichtbar machen. Typische Merkmale wären hier Wirbel und Turbulenz oder Totwasser- und Rückstromgebiete.

Mittels Visualisierungstechnik von Simulationsergebnissen wird Wissen durch Bilder produziert und organisiert, welches mit bloßem Auge nicht erwerbbar ist. McCormick, De Fanti und Brown definieren in ihrem Paper „Visualization in scienfific computing Visualisierung“daher auch als eine Berechnungsmethode, die symbolische Informationen in eine geometrisch-visuelle Beschreibungsform überführt und dadurch das Verständnis und die Kommunikation von Modellen, Konzepten und Daten in der Wissenschaft ermöglicht oder erleichtert. Visualisierung wird dazu verwendet, eine visuelle Repräsentation bereitzustellen, die den Anwender seine Daten nicht nur visuell betrachten lassen, sondern

ihm Hilfsmittel an die Hand gibt, verborgene Zusammenhänge in den Daten aufzudecken und forschungsrelevante Informationen zu extrahieren.
Die Visualisierung wird somit auf drei Stufen eingesetzt:

- die explorative Analyse
- die konfirmative Analyse und
- die Präsentation und Kommunikation

Bei der *explorativen Analyse* gibt es noch keine Hypothese über die vorliegenden Daten bzw. keine Theorie, die die Struktur oder die Eigenschaften des untersuchten Phänomens erklären kann. Mit dieser Art von Visualisierung findet oft eine ungerichtete Suche nach Informationen und Mustern statt. Die Visualisierung kann also im Entstehungskontext des wissenschaftlichen Forschungsprozesses eine Rolle spielen, indem sie einerseits Hinweise zur Formulierung einer Hypothese gibt. Andererseits vermag sie vielleicht sogar neue Theorien oder Forschungsrichtungen anregen, indem Visualisierungen die Modellbildung entscheidend voranbringen, insbesondere dann, wenn Modelle in Form bildhafter Darstellungen auftreten. So wäre z.B. die Entwicklung der Genetik kaum ohne Bilder, wie dem Modell der Doppelhelix und den Röntgeninterferenzbildern denkbar gewesen.
In der *konfirmativen Analyse* liegt eine Hypothese bereits vor, die mit einer geeigneten Visualisierung überprüft und verifiziert werden soll. Es geht hier nicht mehr nur darum, für eine Datenmenge eine geeignete visuelle Repräsentation zu erzeugen. Vielmehr verbindet man mit der Visualisierung eine stärkere erkenntnistheoretische Funktion: Mit ihr sollen Erkenntnisansprüche gerechtfertigt werden. In dieser Hinsicht spielt die Visualisierung im Begründungskontext von wissenschaftlichen Argumentationen (oder sogar Theorien) eine Rolle. Man würde also nicht bloß einen Beleg als empirische Basis austauschen, wenn man eine Visualisierung kommuniziert, sondern diese würde im argumentativen Kontext die notwendigen Informationen zum Verständnis von Prämissen und Konklusion liefern.
Als letzten Schritt kommt es zur *Präsentation und Kommunikation* der erzielten Ergebnisse. Idealerweise liegen zu diesem Zeitpunkt bereits durch die Visualisierung bestätigten Hypothesen vor. Eine Visualisierung sollte die Fakten so darstellen, dass Dritte diese ohne Probleme erfassen und verstehen können.

Hierfür müssen eine Reihe von Anforderungen, insbesondere hinsichtlich der Qualität der Visualisierung beachtet werden. Die Qualität einer visuellen Repräsentation wird im Wesentlichen durch drei Kriterien bestimmt: Expressivität, Effektivität und Angemessenheit.
Grundvoraussetzung einer Visualisierung sollte sein, dass nur die in den Daten enthaltenen Informationen und nur diese dargestellt werden. Diese Fähigkeit einer Visualisierung wird *Expressivität* genannt. Die Einhaltung der Expressivität garantiert eine unverfälschte Vermittlung der für die Forschungsfrage relevanten Aspekte der Daten an den Betrachter. So wird auch das Risiko einer Fehlinterpretation und damit die Gefahr falsche Schlussfolgerungen reduziert.
Eine Visualisierung wird als *effektiv* bezeichnet, wenn sie die visuellen Fähigkeiten eines Betrachters und die Eigenschaften des Ausgabemediums unter Berücksichtigung der

jeweils gewählten Fragestellung und Zielsetzung optimal ausnutzt. Das Effektivitätskriterium gibt Aufschluss über die Fähigkeit einer Darstellung, die in ihr enthaltenen Informationen zu veranschaulichen und auf intuitive Weise an den Betrachter zu vermitteln. Im Allgemeinen sucht man daher nach der effektivsten Darstellungsform für eine gegebene Datenmenge. Hingegen ist jedoch nicht immer die effektivste Visualisierung auch die beste Wahl.
Diese beiden Kriterien stellen sicher, dass alle notwendigen Faktoren gefunden werden, um eine effektive visuelle Repräsentation für ein gegebenes Bearbeitungsziel zu finden. Die Visualisierung von Daten, insbesondere diejenige die man mit Hochleistungsrechner erstellt, ist auch mit Kosten verbunden. Die *Angemessenheit* setzt daher Kosten und Nutzen des Visualisierungsprozesses zur Erreichung der Bearbeitungsziele ins Verhältnis. Hierbei zählen nicht nur Technologiekosten, sondern auch zeitlicher, kognitiver, physischer und psychischer Aufwand für den Betrachter. Unangemessene Visualisierungen können nicht effektiv sein.

4.1 Von der Zahl zum Bild

Ziel des Visualisierungsprozesses ist es, gegebene, u.U. sehr große Datenmengen in Form von Bildern zu repräsentieren, um mittels dieser Darstellungsform Einsichten zu gewinnen, die durch das bloße Studium der Datenmengen nicht hervorgehen. Je nach Fragestellung kann die Visualisierung des gleichen Datensatzes sehr unterschiedlich aussehen, je nachdem wie die Eingangsdaten aufbereitet werden. Deshalb ist es wichtig, sich vorher Gedanken dazu zu machen, welchen Aspekt der gegebenen Daten die Visualisierung zum Vorschein bringen soll.

Die Visualisierung lässt sich in mehrere Zwischenschritte gliedern, die sich in einer so genannten *Visualisierungspipeline* anordnen lassen. Die Abbildung 3.16 illustriert die einzelnen Schritte, die zu einer Visualisierung führen.

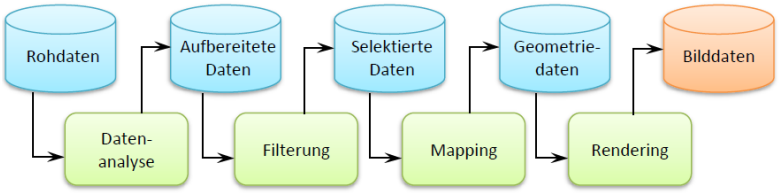


Abbildung 3.16: Die einzelnen Schritte der Visualisierungspipeline

Der Prozess gliedert sich in mehrere Teilschritte: Der Import der sogenannten *Rohdaten*, der Datenaufbereitung und das Filtern der Daten, das Mapping (Erzeugung eines Geometriemodells, d.h. die Abbildung von Datenwerte auf geometrische Primitive) und die Bildgenerierung (Rendering).

Die Rohdaten unterscheiden sich, je nach dem welche Phänomene sie repräsentieren (medizinische Aufnahmen, Strömungen, Deformation in Materialien, statistische Erhebungen, Interaktionen zwischen Menschen,...), in ihrer Struktur. Ausschlaggebend für die Struktur sind Unterschiede im Datentyp und wie viele Werte an jedem Gitterpunkt sich befinden (Dimension des Vektors, in der die Daten pro Gitterpunkt gelistet sind):

Datentyp:

- Skalare Größen: Dies sind numerische Werte, also Zahlen (auch Skalare genannt), die als Werte einer Funktion auftreten können. So kann z.B. die Temperatur an jeder Stelle des Raumes mit einzelnen skalaren Werten hinreichend gut beschrieben werden.
- Vektorielle Größen: Viele Phänomene in der Wissenschaft können durch einzelne Zahl nicht hinreichend beschrieben werden. Sobald Richtungsinformationen von Bedeutung sind, werden Vektoren (oder Tensoren) verwendet. Während für die Beschreibung von Temperatur die Angabe einzelner Werte ausreicht, werden z.B. für die Beschreibung der Luftströmung Vektoren benötigt. Vektoren sind meistens zwei- oder dreidimensional und geben gerichtete Größen oder Positionen an, wie z.B.
  - Position im Raum
  - Bewegungsrichtung
  - Kraft
  - Gradient einer skalaren Funktion („Richtung der Steigung“)
  - Normale zu einer Oberfläche
- Tensorwerte: In komplexeren Fällen, in denen nicht nur eine einzige Richtung, sondern eine Richtungsverteilung gegeben ist, reicht ein Vektor als Richtungsangabe nicht aus. In diesen Fällen werden Tensoren verwendet, die eine Verallgemeinerung von Skalaren, Vektoren und Matrizen auf höhere Dimensionen darstellen. Tensoren werden häufig bei der Visualisierung von Spannungszuständen von Materialien oder der Feldstärke zur Beschreibung elektromagnetischer Felder in der Raumzeit verwendet. Weil Tensoren in diesem Buch keine große Rolle spielen, wird auf diesen Datentyp hier nicht weiter eingegangen.

Dimensionalität: Sie beschreibt die Anzahl der Werte pro Beobachtungspunkt. Der einfachste Fall ist, wenn es für jeden Beobachtungspunkt (Ort und Zeitangabe) genau ein Merkmal gibt, z.B. wenn bei einem Fluid an jedem Gitterpunkt (Ort- und Zeitkoordinate) die (nur) die Geschwindigkeit angegeben wird. Häufig werden aber pro Gitterpunkt mehrere Werte erhoben, z.B. möchte man auch die Bewegung bzw. die Richtung der Strömung und zusätzlich noch die Dichte angeben.

Als nächster Schritt in der Visualisierungspipeline folgt nun die *Datenaufbereitung („Filtering“)*. Dort werden die zur Verfügung stehenden Rohdaten aufbereitet. Es wird nicht

immer möglich sein, stets alle Eigenschaften der Rohdaten akkurat in einer Visualisierung zu beschreiben. Gleichzeitig eine Datenvisualisierung zu finden, die mehreren unterschiedlichen Fragestellungen gerecht werden möchte, ist zum Scheitern verurteilt, da eine solch heterogene Abbildung nicht alle Struktureigenschaften in einem Bild veranschaulichen kann.
Zu den Techniken der Datenaufbereitung gehören die Vervollständigung von fehlenden Datenwerten durch Interpolation. Außerdem müssen Datensätze oft reduziert werden, da in aufwendigen Simulationsstudien oft sehr viele Variablen erfasst werden, die nicht immer für eine gewählte Forschungsfrage von Bedeutung sind und dann vernachlässigt werden können. Eine Datenmenge lässt sich ebenso durch Projektion und Selektion reduzieren. Bei einer Projektion verändert man die Struktur der Datenmenge. Dies bedeutet, dass die Anzahl der Beobachtungspunkte im Raum gleich bleibt, aber nicht alle zugehörigen abhängingen Variablen pro Beobachtungspunkt berücksichtigt werden.
Bei der Selektion einer Datenmenge wird die Anzahl der Datensätze reduziert. Die Selektion wird oft mithilfe eines Filters festgelegt, der gewisse Bedingungen (z.B. einen Schwellenwert) an die Wertebereiche der abhängigen Variablen definiert. Die Definition des Filters kann sich in manchen Fällen direkt aus der Anwendung ergeben, weil auf diese Weise die gewünschten Eigenschaften der Datenmenge erfasst werden. Für spezifische Fragen müssen die Daten oft auch transfomiert werden. Möchte man zum Beispiel etwas über Extrema (maximale Ausprägungen von Variablen), Gradienten oder Wirbel in einer Strömungssimulation erfahren, müssen zunächst erste und zweite Ableitungen der entsprechenden Funktionen berechnet werden.

Das sogennante *„Mapping“* ist der Hauptschritt in der Visualisierungspipeline. Die nun aufbereiteten Daten, die meist nicht-geometrischer Natur sind, werden nun gemäß den Bearbeitungszielen so auf graphische Elemente (Punkte, Linien, Flächen oder Körper) und ihre Attribute wie z.B. Farbe oder Textur abgebildet, dass die in den Daten enthaltenen Strukturen und Zusammenhänge erkannt werden können.
Bei zweidimensionalen Darstellungen gibt es sieben verschiedene visuelle Variablen, d.h. sieben graphische Elemente und Attribute, die für die graphische Darstellung eingesetzt werden können:

- die räumliche Lage mit zwei Richtungskoordinaten
- die Größe
- der Helligkeitswert
- die Musterung oder Textur
- die Farbe
- die Richtung oder Orientierung
- die Form der Elemente.

Für den zwei- und dreidimensionalen Fall sind Punkte, Linien, Vektoren und Polygone (Flächen, meist aus Dreiecken aufgebaut) die graphischen Basisprimitive. Weitere kompliziertere graphische Elemente lassen sich mithilfe dieser darstellen.
Bei der abschließenden *Bildgenerierung (Rendering)* werden die Geometriedaten und ihre Merkmalsbeschreibung in ein digitales Bild (zwei- oder dreidimensional) umgewandelt. Zur Erzeugung dieser Bilder stehen dem Benutzer eine Reihe von Software-Paketen zur Verfügung.

4.2 Einfluss auf die Simulationsmodellierung

Die Darstellung und das Verständnis mathematischer Modelle kann auf verschiedene Weisen erfolgen. Dank der tiefgreifenden Veränderung, die der Gebrauch des Computers als mathematisches Instrument in der Wissenschaft und Technik bewirkt hat, stellt die Visualisierung von Datenstrukturen ein wichtiges Merkmal von Computersimulationen dar: Die visuellen Fähigkeiten der menschlichen Wahrnehmung können in die Modellierung direkt mit einbezogen werden. Genauer formuliert, gilt es die Frage zu beantworten, inwiefern Visualisierung als ein charakteristisches Merkmal von Computersimulationen aufgefasst werden kann. Hier wird infolge die Ansicht vertreten, dass man die Rolle der Visualisierung in der Simulationsmodellierung dadurch bestimmen kann, indem man zeigt, wie Visualisierung in den Modellierungsprozess eingebunden ist und wie sie Möglichkeiten zur Interaktion mit Modellen bietet. Die These wird sein, dass man über die Visualisierung einen neuen methodischen Zugang zu bis dato sehr komplexen und undurchschaubaren Modellen erhält, obwohl die Modelle selbst epistemisch opak bleiben. Das bedeutet, dass die Visualisierung wie eine Rückkopplungsschleife zwischen Modellverhalten und dessen Anpassung durch den Modellierer fungiert: Die Visualisierung erlaubt Aussagen zur Passform des Modells ohne dabei von der Einsicht in die mathematisch (undurchschaubare) formale Darstellung des Modells abzuhängen und auf eine theoretische Durchdringung der Wirkverhältnisse angewiesen zu sein.

Wie Visualisierungen in die Modellierung eingebunden sind und Einblick geben in komplexe Modelldynamiken wird hier anhand eines chaotischen dynamischen Systems verdeutlicht und an einem Beispiel erläutert, nämlich an der Simulation des Hurrikans „Opal“, der sich 1995 über dem Gold von Mexiko bildete.
Ein *chaotisches dynamisches System* ist ein System von partiellen DGL'en. Das Verhalten solcher Systeme ist einer bestimmten Hinsicht sehr komplex. Es handelt sich zwar um ein deterministisches System, bei dem durch die Festlegung der Anfangsbedingungen das Verhalten des Systems vollständig festgelegt ist. Gleichzeitig ist jedoch die zukünftige Entwicklung des Systems unvorhersagbar oder „chaotisch“, in dem Sinne, dass die kleinste Modifikation der Anfangsbedingungen zu beträchtlichen Veränderungen im Gesamtverhalten des dynamischen Systems führt.
Dieses Modell trat zum ersten Mal bei dem Meteorologen und Physiker Edward Lorenz auf, der im Bereich meteorologischer Modellierung gearbeitet und als erster die Rätselhaftigkeit dieses chaotischen Verhaltens beobachtet hatte. Seine Erkenntnisse zu

der empfindlichen Abhängigkeit von den Anfangsbedingungen in einem dynamischen System im Jahr 1963 begründeten die darauf aufbauende Chaostheorie und lieferten die Grundlagen zur Formulierung einer Theorie komplexer dynamischer Systeme. Die nicht vorhersagbare Systemdynamik, die bei der gerinsten Variation der Anfangsbedingungen ensteht, wurde später als der so genannter Schmetterlingseffekt bekannt.

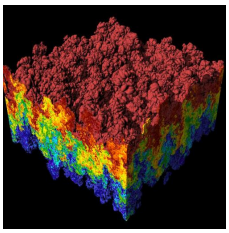
Dynamische Systeme sind in der Sprache der mathematischen DGL'en formuliert. Mit traditionellen Methoden der Differentialrechnung und analytischen Lösungsverfahren von DGL'en konnte man zwar das rätselhafte Verhalten dieser dynamischen Systeme feststellen, jedoch versagten diese mathematischen Instrumente bei der Analyse solch komplexer Systeme und lieferten somit kein Verständnis, wie es zu diesem unvorhersagbaren Verhalten in einen chaotischen Systemen kam.
Mit der Computersimulation und insbesondere der Visualisierung wurde ein alternatives Instrument verfügbar, das zu einem Verständnis dieser undurchschaubaren und komplexen Modelle verhalf. Denn tatsächlich war es so, dass erst die visuelle Darstellung der Endzustände, auf die sich ein dynamisches System im Laufe der Zeit zubewegt und die unter der Dynamik des Systems nicht mehr verlassen werden, die entscheidenden Erkenntnisse für die Entwicklung einer Theorie komplexer dynamischer Systeme gegeben hatte. Nach der Implementierung der partiellen DGL'en auf dem Computer konnte der Rechner zu jedem hypothetischen Anfangszustand den nächsten Zustand des Systems (numerisch approximativ) berechnen und die isolierten Einzelergebnisse der Rechnungen zu einem Bild zusammenfügen, sodass dadurch ein anschaulicher Eindruck der zeitlichen Entwicklung des Systems erfahrbar wurde.
Wichtig ist also festzuhalten, dass das Verständnis chaotischer Systeme stark auf einer visuellen Einsicht von (computergenerierten) Bildern basierte. Es beruht also nicht (vollständig) auf der verstandesmäßigen Durchdringung einer grundsätzlich durchschaubaren formalen Darstellung, sondern auf der Kombination von undurchschaubarer formaler Darstellung (des dynamischen Systems), der Implementierung des Systems auf dem Computer und die Transformation der einzelnen Ergebnisse in ein visuelles Bild. Die Computervisualisierung macht solche Systeme somit zuerst einmal für unsere Wahrnehmung zugänglich und eröffnet damit einen explorativen Umgang mit Visualisierungen, indem mit ihnen Aufschluss über den Verlauf möglicher Endzustände des dynamischen Systems gewonnen und komplexes Verhalten anschaulich fassbar wird.
Die Rolle der Visualisierung soll nun anhand des Hurrikan Beispiels weiter verdeutlicht werden. Im Herbst im Jahr 1995 zog der auf „Opal“ getaufte Hurrikan über den Golf von Mexiko auf das amerikanische Festland zu und richtete wüste Zerstörungen mit Todesfällen und Sachschäden von mehreren Milliarden Dollar an. Die zeitnahe Vorhersage des Verlaufs von Hurrikans ist zu einem wichtigen Forschungsgebiet der angewandten Meteorolgie geworden. In diesem Zusammenhang initiierten das US-amerikanische National Center for Supercomputing Applications (NCSA) und das Department of Atmospheric Sciences der Universität von Illinois ein Projekt, mit dem Ziel die Dynamik von Hurrikans zu erforschen und durch die Analyse und Rekonstruktion von Opal ein Modell zu entwickeln, das eine Vorhersage des Weges, der Geschwindigkeit und des Niederschlages für zukünftige Stürme möglich machen sollte. Eine Reihe wichtiger Daten zu Opal waren im

Nachgang des Hurrikans bekannt, sodass ein Simulationsmodell konstruiert wurde, das den tatsächlichen Verlauf von Opal möglichst gut rekonstruieren sollte. Durch Variation der Anfangsbedingungen sollten dann weitere Stürme für Vorhersagen simuliert werden können.
Als Grundlagen dieses Modells dienten allgemein physikalische (hydrodynamische) Gesetze, die als ein System von partiellen DGL'en formuliert sind. Die tatsächliche Anpassung des Modells an das Verhalten des Hurrikans Opal erfolgte über ein iteratives Verfahren, das von der Folge Computerexperiment, Beobachtung der Visualisierung, Abgleich mit der Dynamik von Opal und Neujustierung von Parametern bestimmt war: Die Forscher beobachteten den simulierten Verlauf des Hurrikans, änderten dann verschiedene Parameter so lange ab, bis der simulierte dem tatsächlichen Verlauf von Opal aufgrund von visueller Übereinstimmung ausreichend nahe kam.
Die mit den Parameteränderungen verbundenen Zusatzannahmen wurden oft nicht durch theoretische Erwägungen begründet. Als Begründung diente hauptsächlich die durch die Visualisierung erzeugte Anschaulichkeit der simulierten Hurrikan-Verläufe. Um das Modell anzupassen, erwies sich die Neujustierung gerade derjenigen Parameter mit unklarer physikalischer Interpretation als erfolgreicher als diejenigen Parameter die auf physikalischen Annahmen beruhten.
Die Visualisierung ermöglichte also einen Zugang zum Modellverhalten, der andernfalls erkenntnistheoretisch unzugänglich geblieben wäre.

4.3 Techniken der Visualisierung

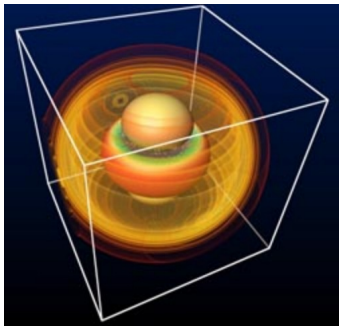
Das Ziel jeder Visualisierung ist es, die aufbereiteten Daten so auf geometrische Elemente abzubilden, dass die in den Daten enthaltenen Strukturen und Zusammenhänge erkannt werden können. Das Mapping, d.h. die geeignete Wahl der graphischen Elemente und Attribute, stellt den Kernprozess der Visualisierung dar. Je nach Datentyp, die im Abschnitt 4.1 spezifiziert worden sind, kann man die Techniken der Visualisierung anhand des Datentyps klassifizieren.

- Algorithmen für Skalardaten.
  Für räumlich strukturierte Skalardaten gibt es eine Reihe von Visualisierungstechniken, so z.B.
  - Einfärben (Colormapping) für 2D Daten: Die darzustellenden Geometrie wird entsprechend den gegebenen Skalarwerten eingefärbt.
  - Schnittfläche mit Einfärben für 3D Daten: Hier wird der dreidimensionale Raum mit einer zweidimensionalen Fläche (meist eine Ebene) geschnitten und diese dann mittels Colormapping eingefärbt.
  - Volumenvisualisierung: Isolinien (2D) und Isoflächen (3D): Isolinien und -flächen zeigen nur jene Punkte eines 2D-/3D-Volumendatensatzes an, die einem bestimmten Skalarwert entsprechen: Marching Squares und Marching Cubes-Algorithmus.
  - Volumenvisualisierung: Direktes Volumenrendering für 3D Daten: Hier wird der gesamte Volumendatensatz visuell dargestellt. Die Skalarwerte an jedem Gitterpunkt werden auf das Bild projiziert. Jedem Skalarwert werden optische Eigenschaften wie Absorption oder Emission zugewiesen. Anschließend werden Sichtstrahlen für jeden Pixel des Bildes durch das Volumen verfolgt, die berechnen welche Farbe der Betrachter im jeweiligen Pixel sieht.
- Visualisierung von Vektordaten. Eine typische Quelle für Vektorfelder sind Anwendungen aus der Numerischen Strömungsmechanik, wobei dort Vektoren die Strömungsrichtung eines Fluids beschreiben. Eine Standardmethode um diese zu visualisieren ist der Einsatz von Stromlinien (oder Röhren), die die lokale Strömungsrichtung darstellen.
- Visualisierung von Tensordaten: Hier handelt es sich um Algorithmen, die die Richtungsverteilung bestimmter Größen beschreiben. So kann man z.B. Materialeigenschaften wie Spannung, Dehnung oder anisotrope elektrische Leitfähigkeit beschreiben.

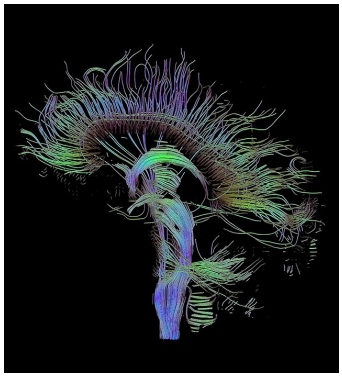


(a) Colormapping. Die Farben zeigen eine Zwei-Phasen-Instabilität an, die eine Störung an der Grenzfläche zweier unterschiedlich dichter Flüssigkeiten exponentiell wachsen lässt.

(b) Volumenvisualisierung: Visualisierung eines Schädels.



(c) Visualisierung von Vektordaten: Mehrere Supercomputer simulieren die Gravitationseffekte von Kollisionen schwarzer Löcher



(d) Visualisierung von Tensordaten: Bildgebung einer diffusionsgewichteten Magnetresonanztomographie, die die Diffusionsbewegung von Wassermolekülen in Körpergewebe, hier im Gehirn, misst und räumlich aufgelöst darstellt. Bei Erkrankungen des zentralen Nervensystems erlaubt die Richtungsabhängigkeit der Diffusion Rückschlüsse auf den Verlauf der großen Nervenfaserbündel.

4.4 Fokus: Visualisierung von Volumendaten

Bei der Volumenvisualisierung möchte man wichtige Eigenschaften, die sich in dreidimensionalen Daten befinden, visuell sichtbar machen, um Zusammenhänge aufzudecken, die durch das Betrachen von Zahlenkolonnen nicht erfassbar sind. Im Folgenden wird detaillierter auf die Visualisierung von Volumendaten (Skalardaten) eingegangen, genauer wird hier die Idee des Marching Cube Algorithmus vorgestellt, mit dem die Oberfläche von dreidimensionalen Objekten durch das Aneinanderlegen von vielen Dreiecken approximiert wird. Ebenso werden Vor- und Nachteile im Vergleich mit dem direkten Volume Rendering aufgezeigt, bei dem statt nur der Oberfläche von 3D Daten alle dreidimensionalen Informationen auf ein 2D Bild auf dem Computer abgebildet werden.

4.5 Begrifflichkeiten

Volumendaten bestehen aus einer Menge V von Tupeln (x, y, z, v), wobei die ersten drei Werte (x, y, z) die Koordinaten einem dreidimensionalen Bezugssystem anzeigen und den Datensatz D bilden. Der Wert v ist der entsprechendn skalare Wert (z.B. die Dichte, Temperatur, Ladung, ...) an einer Stelle im Bezugssystem. Weiter wird vorausgesetzt, dass die Beobachtungspunkte (x, y, z) auf einem regelmäßigen dreidimensionalen Gitter angeordnet sind. Volumendaten sind also skalare Daten, wobei an jedem Gitterpunkt genau ein skalarer Wert gegeben ist. Ein Datum (x, y, z, v) wird auch als *Voxel* (in Analogie zu Pixel) bezeichnet. Die skalaren Daten sind oft Funktionswerte einer mathematischen Funktion $f(x, y, z)$, die z.B. die Dichte, Temperatur, Ladung,... an jedem Punkt (x, y, z) auf den Gitter berechnet. Das Gebiet mit den Volumendaten wird als *Datenwürfel* bezeichnet. Da Volumendaten ein regelmässiges Gitter voraussetzen, ergeben sich die Koordinaten der Gitterpunkte implizit und müssen nicht explizit gespeichert werden. Ein Datenwürfel besteht also aus vielen Gitterzellen mit ensprechenden Skalarwerten auf den Gitternetzpunkten. Acht benachbarte Voxel bilden ein *Volumenelement* oder eine *Zelle*.

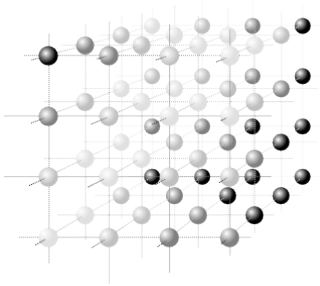
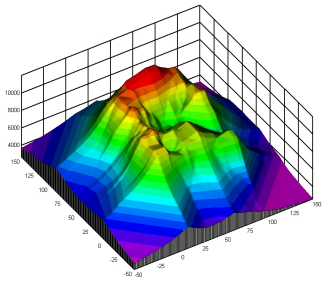


Abbildung 3.18: Dies ist die bildliche Veranschaulichung eines Datenwürfels. Jeder Voxel erhält abhängig von seinem Wert eine unterschiedliche Graufärbung.

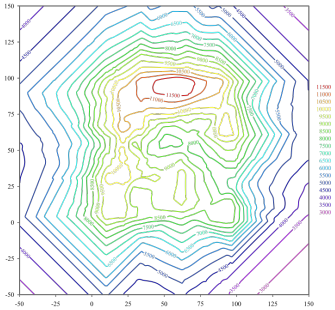
Skalarwerte zwischen Gitternetzpunkten sind oft ebenfalls von Interesse. Die Skalarwerte auf den Kanten und innerhalb eines Volumenelements werden durch eine sogenannte

trilineare Interpolation bestimmt. Dies ist eine Technik bei der die bekannten acht Eckpunkte und die zugehörigen acht Skalarwerte eines Volumenelementes benutzt werden, um einen Zwischenwert innerhalb des Volumenelementes zu approximieren.
Um räumliche Strukturen von Volumendaten zu gewinnen, verwendet man Isolinien (2D) oder Isoflächen (3D), die genau jene Punkte im Datenwürfel verbinden, die den gleichen Skalarwert haben.
Eine *Isolinie* I ist also definiert als die Menge aller Punkte $P(x, y, z)$ eines Datensatzes, die den gleichen Skalarwert v haben. Der Wert v ist oft Funktionswert einer Funktion f, die jedem Beobachtungspunkt P einen Wert v zuweist. Die Definition einer Isolinie $I(v)$ lautet daher:

$$I(v) = \{ \text{ Menge aller Punkte } P(x, y, z) \text{ in } D \text{ sodass } f(x, y, z) = v\}$$



(a) Hier ist eine dreidimensionale Oberfläche abgetragen.



(b) Passend zur Oberfläche links sieht man hier die entsprechenden Iso- bzw. Höhenlinien.

Im zweidimensionalen ergeben sich Isolinien, die vergleichbar mit Höhenlinien auf einer Landkarte sind. Höhenlinien geben direkt den räumlichen Verlauf einer Größe, z.B. der Höhe in Meter wider. Im dreidimensionalen spricht man dann von *Isoflächen*. Mithilfe von Isoflächen kann man geometrische und topologische Eigenschaften von Wertverteilungen aufzeigen. In der Medizin zum Beispiel werden sie verwendet, um aus Datensätzen mit Dichtewerten Organoberflächen, Knochenbau oder Gefäßsysteme darzustellen. Die Daten werden hier meist über bildgebende Verfahren (CT, MRT) gewonnen. Außerdem können mit der Oberflächenvisualisierung auch Flächeninhalt und Volumen von Geschwulsten abgeschätzt oder Prothesen modelliert werden.
Um die Zuordnung von Datenpunkten zu einer Isofläche vornehmen zu können, muss selbstverständlich der skalare Wert v im Voraus bekannt sein. Die Festlegung dieses so genannten Schwellenwerts ist nicht immer einfach und erfordert ein gewisses Maß an Vertrauheit mit den Daten. Alle Daten, die nicht dem Schwellenwert entsprechen, werden nämlich auch nicht visuell dargestellt. Ist der Schwellenwert also falsch gewählt, kann es passieren, dass relevante Informationen ausgeblendet werden. Ebenso kann es passieren, dass durch die Angabe eines eine ungeeigneten Schwellenwerts Bilder entstehen, die falsche Schlussfolgerungen suggerieren. Aus diesem Grund wird oft gerne mit Schwellenwertintervallen (durch Angabe einer Ober- und Untergrenze) statt mit einem einzelnen Schwellenwert gearbeitet. Ober- und Untergrenze ergeben sich meist nicht aus den Daten selbst, sondern über Hintergrundwissen und eigener Erfahrung im entsprechenden Arbeitsgebiet gefunden werden. Bei häufiger und erfolgreicher Anwendung der Volumenvisualisierung, z.B. in der Medizin, etablieren und bewähren sich über die Zeit in der Regel bestimmte Schwellenwerte zur Unterscheidung von Gewebetypen.

Der Datenumfang von Volumendaten ist in der Regel sehr groß. Für einen Datenwürfel von nur $64 \cdot 64 \cdot 64$ Voxel benötigt man eine Viertel Million Speicherplätze. Bei vier Bytes pro Voxel entspricht das bereits einem Megabyte an Daten. Klassische Volumenvisualisierungen in der Medizin arbeiten typischerweise mit deutlich höheren Auflösungen. Moderne Computertomographen schaffen seit 2004 eine gleichzeitige Aufnahme von 64 parallelen Schichten. Eine Schicht mit $512 \cdot 512$ Volumenelementen entsprechen bereits 40 Megabyte

Speicher. Zur Visualisierung sind deswegen hochleistungsfähige Rechner vonnöten. Wegen der Datengröße sind in der Regel bisher noch keine interaktiven Manipulationen möglich.

4.6 Marching Cubes Algorithmus

Die grundlegende Idee des Marching Cubes Algorithmus besteht darin, Flächen (Isoflächen) aus dem Datensatz zu generieren, um auf diese Weise die Oberfläche von Objekten zu approximieren. Die grundlegenden Schritte der Volumenvisualisierung entsprechen der Visualisierungspipeline, wie sie in Abschnitt 4.1 vorgestellt wurden.

Nach Erhalt der Volumendaten, z.B. von bildgebenden Verfahren, erfolgt ggf. in der Datenaufbereitung (Filtering) eine Datenvervollständigung oder Datenreduktion. Nun werden die skalaren Daten in einem räumlichen Bezugsystem ausgewählt, die als Gitterpunkte eines regelmäßigen Gitters vorkommen.
Im Mapping-Schritt müssen die Datenwerte an den Gitterpunkten auf graphisch-visuelle Attribute abgebildet werden. Diese Werte resultieren oft aus Messungen und repräsentieren physikalische Eigenschaften. Gesucht ist nun also ein Mapping von Voxelwerten auf optische Eigenschaften (Farbe, Opazität/Transparenz). Um nun Strukturmerkmale in den Daten zu identifizieren, muss eine Klassifikation in Datenklassen durchgeführt werden. Voxel mit ähnlichen Skalarwerten sollen sich in der gleichen Datenklasse befinden, sodass Datenwerte, die unterschiedliche Eigenschaften repräsentieren, auch in der Visualisierung unterscheidbar sind. Den einzelnen Klassen werden dann Farb- und Transparenzeigenschaften zugeordnet. Die Einführung von Datenklassen ist ein sensibler Prozess und potenziell fehleranfällig. In jedem Fall ist die Klassifikation anwendungsabhängig. Schon eine leichte Änderung des Klassifikationskriteriums kann zu deutlich anderen Ergebnissen führen.
Ein individuelles Mapping für jeden einzelnen Voxel wird nicht verfolgt. Eine intuitive Idee einer solchen Datenvisualisierung wäre die Dekomposition der Datenmenge in einzelne Datenpunkte, die dann im Mapping jeweils einzeln mit Farb- und Transparenzeigenschaften versehen werden. Damit stößt man aber bei der Interpretation des Bildes auf Schwierigkeiten. Man möchte ja schließlich in der visuellen Analyse gewisse Sturktureigenschaften in den Daten erkennen können, die womöglich in einer großen Menge von verschiedenfarbigen Bildpunkten nicht ausreichend zur Anschauung gebracht werden können. Stattdessen gibt man sich als Kriterium für die Partition in Datenklassen nicht einen Wert, sondern ein Intervall an Werten vor. Jedem Datenwert ordnet man dann eine Zugehörigkeitswahrscheinlichkeit zu einer Datenklasse zu. Die dabei entstehenden Wahrscheinlichkeiten werden dann in der Visualisierung bzw. im Mapping berücksichtigt.

Eine bessere Strategie zur Volumenvisualisierung besteht darin, Flächen (i.d.R. Isoflächen) aus der Datenmenge zu extrahieren. Isoflächen sind Flächen, die, wie weiter oben erklärt, all jene Voxel enthalten, deren Skalarwert dem Schwellenwert der Isofläche entsprechen. Mithilfe dieser Isoflächen erfasst man leichter geometrische und topologische Eigenschaften der Wertverteilungen der Daten. Ebenso können damit einfach Flächeninhalte oder

Volumina abgeschätzt werden. Im Rendering-Schritt werden die ermittelten Isoflächen verarbeitet und graphisch in einem Bild dargestellt. Hier handelt es sich also um eine indirekte Volumenvisualisierung, weil nur die Oberfläche von Objekten visualisiert werden.

Der sogenannte *Marching Cubes Algorithmus* von Lorensen und Cline (1987) ist das bekannteste Verfahren zur Modellierung von Oberflächen von 3D Daten. Man nährt die Oberfläche durch ein polygonales „Drahtgittermodell“ an, indem man eckige Flächen (in der Regel Dreiecke) so anordnet, dass sie die Oberfläche des Objektes nachbilden.
Die Volumendaten eines Objekts wurden bereits im Datenwürfel in Zellen/Volumenelemente (cube) zerlegt, die jeweils aus acht Voxeln bestanden. Die grundlegende Idee von Marching Cubes besteht nun darin, dass durch die einzelnen Zellen zu „wandern“, um zu sehen, welche Seiten der Zellen innerhalb und welche außerhalb der Isofläche liegen um daraufhin die Isofläche durch Dreiecke zu approximieren.

Das Vorgehen im Detail wird nun im Folgenden beschrieben:
Zuerst muss der Schwellenwert v bekannt sein, der die gesuchte Isofläche eindeutig festlegt. Nun wandert man von Zelle zu Zelle. In jeder Zelle bestimmt man nun, welche der acht Voxel (die Ecken einer Zelle) innerhalb und außerhalb der Isofläche liegen. Hierfür seien die Eckpunkte der jeweiligen Zellen mit $V1, \ldots, V8$ bezeichnet und die Datenwerte an den Ecken entsprechend mit $W1, \ldots, W8$ (vgl. Abbildung 3.20). Ist der Wert an den Eckpunkten größer v, dann wird er als „außen“, andernfalls als „innen“ klassifiziert. Diese Zuordnung kann nun als 8-Bit (1 Byte) Codierung pro Voxelzelle codiert werden. Eine „1“ steht in der Klassifikation für „innen“, eine „0“ für „außen“. Dieses Byte kann dann in einer Tabelle abgespeichert werden.
Die Schnittpunkte der Isofläche mit den Kanten der jeweiligen Zelle werden mithilfe von linearer Interpolation der Daten in den Eckpunkten ermittelt. Diese Schnittpunkte definieren dann die drei Eckpunkte eines Dreiecks (oder Vielecks), das die Isofläche approximiert.

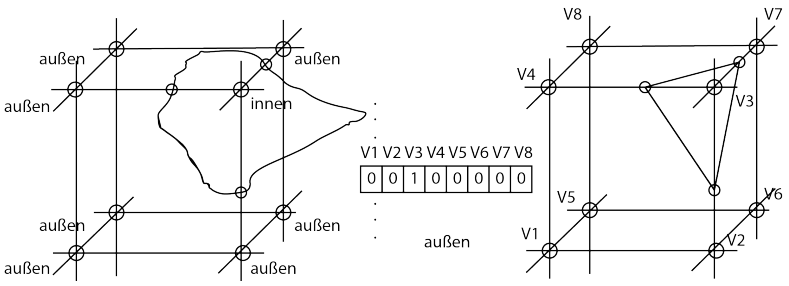


Abbildung 3.20: Links sieht man die aktuelle Zelle, in der die Isofläche angedeutet ist. Nur der Eckpunkt $V3$ liegt innerhalb der von der Isofläche bestimmten Fläche. Rechts im Bild ist stellt das Dreieck eine Approximation an die Isofläche dar.

Theoretisch lassen sich gemäß der Bytebelegung $2^8 = 256$ mögliche Fälle unterscheiden. Aufgrund von Symmetrie kann diese Zahl aber auf 15 verschiedene Fälle reduziert werden. Leider ist die Reduktion auf diese 15 Fälle nicht ausreichend , weil Zweideutigkeiten

beim Verbinden der Schnittpunkte auf den Kanten auftreten können. Und zwar dann, wenn an einer Seitenfläche zwei Eckpunkte als „innen“ und zwei als „außen“ klassifiziert werden, wobei die diagonal gegenüberliegenden Eckpunkte die gleiche Klassifikation nach innen-außen haben. Dieser Fall ist in der folgenden Abbildung 3.21 dargelegt.

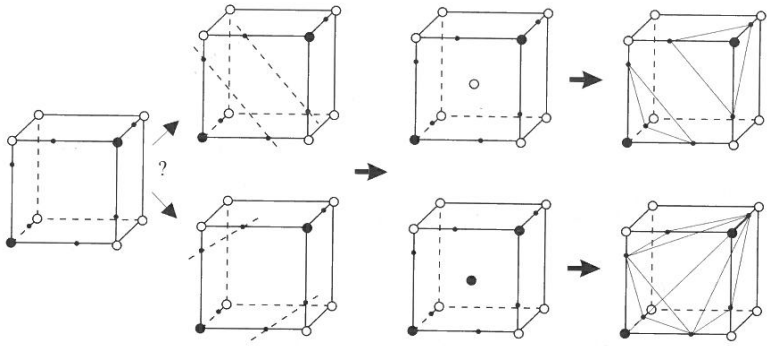


Abbildung 3.21: Die nicht-eindeutige Isoflächen-Approximation im Marching Cube Algorithmus.

In diesem Fall werden alle vier Kanten dieser Seitenfläche geschnitten. Dadurch ist nicht mehr eindeutig, wie die Schnittpunkte zu einem Dreieck (Vieleck) zu verbinden sind. Die weißen Punkte sind diejenigen Eckpunkte, die als „innen“, schwarz die Eckpunkte die als „außen“ klassifiziert sind. Die Idee besteht nun darin, einen weiteren Datenpunkt in der Mitte der Fläche durch Interpolation zu bestimmen und ihn dann auch zu klassifizieren, ob er innen oder außen liegt. Entsprechend können dann die Schnittpunkte auf den Kanten eindeutig verbunden werden (vgl. Abbildung 3.21).
Zusammenfassend stellt die Flächenextraktion über den Marching Cubes Algorithmus eine Technik zur Volumenvisualisierung dar, die die Analyse von geometrischen und topologischen Eigenschaften von Volumendaten im Raum erleichtert. Der Gesamtrechenaufwand des Algorithmus wird bestimmt durch den Aufwand zur Berechnung der Isoflächen-Approximationen. Für jeden Isowert/Schwellenwert müssen alle Zellen durchlaufen werden. Bei hochauflösenden Daten mit feinmaschigen Datenwürfeln müssen also sehr viele Dreiecke erzeugt werden, was zu einem hohen Speicherbedarf und großen Renderzeiten führt. Die Reduzierung der zu durchlaufenden Zellen ist daher ein guter Optimierungsansatz.

Die Extraktion der Oberflächen von Objekten bleibt dennoch ein aufwendiger Prozess. Informationen, die sich innerhalb der Daten befinden, gehen verloren, die z.B. bei einem CT-Scan eines menschlichen Körpers relevant sein können. Ebenso können Dinge wie Nebel, Rauch oder Feuer mit diesem Ansatz schlecht visualisiert werden. Sind tatsächlich Volumendaten über die Oberfläche hinaus von Bedeutung, ist der Ansatz des Marching Cubes Algorithmus nicht geeignet.
Für diese Art von Visualisierung wird das sogenannte *Direkte Volumenrendering* verwendet. Bei der direkten Volumenvisualisierung geht es um die direkte Visualisierung von Volumendatensätzen bzw. Voxeldaten auf einem Gitter. Direkt bedeutet hier, dass

direkt mit den Voxeldaten gearbeitet wird, vorher also nicht erst eine Oberflächen-Approximation angefertigt wird, wie zum Beispiel bei der Erstellung von Isoflächen mit dem Marching Cube Algorithmus. Direktes Volumenrendering vermittelt damit mehr Informationen, allerdings unter Kosten einer höheren Komplexität des Algorithmus und längeren Erstellungszeiten bei der Bildgenerierung.
Eine Möglichkeit für direktes Volumenrendering beruht auf einem Emissions-Absorptionsmodell, das auf der physikalischen Grundlage des Strahlungstransports basiert, der die Interaktion von Licht, Objekten und dem dazwischen liegendem Medium beschreibt. Bei der Interaktion von Licht mit einem Medium finden die folgenden Interaktionen statt:

1. Emission: Material emittiert Licht, strahlt Licht aus
2. Absorbtion: Material absoriert Licht, Licht wird reduziert
3. Streuuung: Die Richtung des Lichts wird verändert

Für das Emissions-Absorptionsmodell in der Computergraphik werden Streuuung und Reflexion durch andere Voxeln nicht berücksichtigt, da dies die Bildgenerierung zu stark verkomplizieren würde. In der Regel sagen die skalaren Werte in den Volumendaten nicht direkt etwas aus über die physikalische Wechselwirkung mit Licht. Diese Werte können also in dieser Form noch nicht für die Bildgenerierung benutzt werden. Deshalb werden jedem Skalarwert durch eine sogenannte Transferfunktion physikalische Größen wie Emission und Absorption zugewiesen. Mit dieser Transferfunktion wird es dann möglich, verschiedene Strukturen innerhalb eines Datensatzes zu unterscheiden. Bei den meisten Verfahren werden jedem skalaren Wert ein Quadrupel RGBA zugeordnet. R, G und B geben die Emission im roten, grünen und blauen Frequenzbereich des Lichts an. Der sogenannte Alpha-Wert (A) gibt die Opazität an und stellt ein Maß für die Absorption bzw. Lichtundurchlässigkeit dar.
Für die visuelle Repräsentation des Volumendatensatzes stellt man sich nun die Emission von Lichtstrahlen vor, die auf dem Weg zum menschlichen Auge auf verschiedene Materialien mit entsprechenden Absorptionswerten trifft. Je nach Hindernissen, auf die ein Lichstrahl trifft auf dem Weg ins Auge, reduzierte sich entsprechend die Lichstärke des Strahls. Da man an der Gesamtmenge an Lichtstrahlen interessiert ist, die auf das Auge treffen, müssen die Beiträge aller Lichtstrahlen aufsummiert werden, was schließlich zum so genannten Volume-Rendering-Integral führt, das von Hand nicht auszurechnen ist.
Das Raycasting-Verfahren ist ein Verfahren, das oben genannte Integral numerisch zu approximieren. Es wird im medizinischen Bereich in der Computertomographie (CT) und Magnetresonanztomographie (MRT), wie auch in der numerischen Simulation der Strömungsmechanik von Gasen und Flüssigkeiten eingesetzt.

Dabei sendet man „Sehstrahlen“ vom Betrachter durch das Zentrum eines Pixels aus und tastet auf dessen Weg das Volumen in gleichmäßigen Abständen ab, um Farb- und Opazitätswerte zu berechnen, wie in der Abbildung 3.22 dargestellt. Das Aufsummieren der abgetasteten Werte ergibt dann die Approximation des Volumenintegrals.

Auch hier muss die Schrittweite, d.h. die Anzahl der Abtastpunkte, denen Farb- und Opazitätswerte zugeteilt werden, mit Vorsicht gewählt werden. Bei einer zu großen

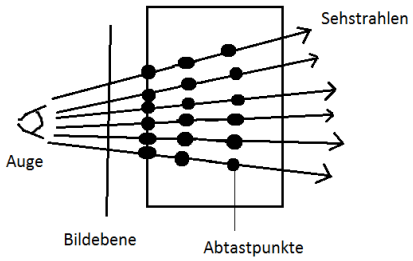


Abbildung 3.22: Für jeden Pixel wird der Farbwert in Abhängigkeit von Absorption und Emissionseigenschaften aus dem Volumen berechnet.

Schrittweite können Wechsel von Struktureigenschaften in den Daten verloren gehen. Bei zu kleiner Schrittweite steigt der Rechenaufwand immer mehr an.

4.7 Grenzen der Visualisierung

Die Anschaulichkeit und der interaktive Zugang zu Modellen macht die Visualisierung zu einer wichtigen Komponente in der Computersimulation. Es ist eine charakterisierende Eigenschaft von Visualisierungen im Rahmen von Computersimulationen, dass sie eine visuelle Einsicht in die Dynamik von komplexem Modellverhalten liefern können. Weil Visualisierungen Möglichkeiten zur Interaktion mit Modellen bieten, ist mit ihnen quasi ein experimenteller Zugang zu Modellen erreicht, sodass einzelne Modellannahmen nach Maßgabe der Performanz des simulierten Systems angespasst werden können, um z.B. empirisch beobachtete Abläufe bestmöglichst nachzubilden.
Forscher „beobachten“ mit der Visualisierung das Verhalten von Simulationsmodellen, indem sie Anfangsbedingungen oder Paramterwerte varriieren, dann die entsprechenden visualisierten Ergebnisse betrachten und Schlüsse ziehen.
Dieses Vorgehen ähnelt methodisch der Arbeit mit einem wissenschaftlichen Experiment, sodass Forschung mithilfe von Simulationen als eine Art wissenschaftliches Experimentieren aufgefasst werden kann. Die Eigenart moderner Naturwissenschaft beruht unter anderem auf dem Experiment. Die experimentelle Vorgehensweise wird sogar häufig mit der naturwissenschaftlichen Methode gleichgesetzt. War in der Antike und Mittelalter „experimentieren“ eher eine Art passives Beoobachten der Natur, wird ab der Neuzeit das aktive Eingreifen in die Natur Wesensmerkmal von empirischer Wissenschaft.
Vergleicht man nun Simulationen mit wissenschaftlichen Experimenten, besteht die Gefahr, simulierte Systeme, die mit Computersimulationen erforscht werden, mit realen Systemen, die mit Experimenten erforscht werden, gleichzusetzen. Mit dieser scheinbaren Nähe von Computersimulation und Experiment ist gleichermaßen auch Reichweite und Grenze von Computervisualisierung markiert.
Durch das Experiment werden Aspekte der Wirklichkeit erfahrbar, die ohne die verwendeten Instrumente im Experiment in der bloßen Natur nicht erkennbar wären. Während das

Instrument also im wissenschaftlichen Experiment u.a. die produktive Funktion hat, die Wirklichkeit des Menschen zu erweitern, also Phänomene zu erzeugen, die sonst nicht im Bereich menschlicher Erfahrung von Welt auftreten, manipulieren wir mit dem Computer als Instrument nicht die Wirklichkeit. Verstärkt durch die außerordentlich anschauliche Qualität von Computervisualisierungen, sollte man daher nicht den vorschnellen Schluss ziehen, dass Visualisierungen stets Entitäten oder Prozesse in der Welt darstellen. Man sollte sich also stets der Aspekthaftigkeit dieser Bildrepräsentationen bewusst sein. Visualisierungen erlauben keine vollständige Beschreibung eines Phänomens der Welt. Visualisierungen sind stets selektiv, abstrahierend und reduzierend.
Eine weitere Problematik lässt sich anhand der Computersimulation von chaotischen dynamischen Systemen beschreiben. Solche Systeme - die in Abschnitt 4.2 zur Sprache kamen, um zu zeigen, wie Visualisierungen Möglichkeiten zur Interaktion mit Modellen bereitstellen - machen einen weiteren skeptischen Moment im Umgang mit Visualisierungen deutlich.
Die Abänderung der Anfangsbedingungen dieser dynamischen Systeme führte zu nicht vorhersagbaren Modelldynamiken. Nehmen wir den Fall an, dass eine Computervisualisierung angemessen das Verhalten eines spezifischen dynamischen Systems beschreibt, dann spielen kleine Abweichungen in den Anfangsbedingungen eine große Rolle. Berücksichtigt man, dass bei der Implementierung des mathematischen Modells auf dem Computer natürlicherweise Fehler entstehen und bedenkt man, dass sich das visuelle Bild selbst durch Datenaufbereitung, gewählte Visualisierungstechnik manipulieren lässt, so ist nicht mehr so klar, worauf die in einer Visualisierung erkennbaren Vorgänge ursächlich zurückzuführen ist: Entspringen sie der tatsächlichen Modelldynamik oder handelt es sich (nur) um instrumentell erzeugte Artefakte?
So kann sich zum Beispiel bei der Simulation einer Flüssigkeitsströmung, die ein starres Hindernis umfließt, hinter dem Hindernis eine spezifische Verwirbelung bilden, die man in der Strömungsvisualisierung beobachten kann. Was erklärt diesen Wirbel? Es kann ein Effekt sein, der sich aus den Modellannahmen begründen lässt oder der Effekt ist ein Resultat von instrumentellen Einstellungen einer bestimmten Implementierung auf dem Computer, z.B. durch die spezielle Wahl eines Gitters bei der Diskretisierung. Die Ursache kann also modellintern gesucht werden oder (nur) in der technisch vermittelten Übersetzung des Modells auf einen Computer begründet liegen. Im letzteren Fall hat man es mit einem instrumentell erzeugten Artefakt zu tun. Im Rahmen einer Validitätsanalyse könnte man überprüfen, ob sich die Verwirbelung auch unter der Wahl von anderen Diskretisierungen (d.h. unter der Wahl feinerer Gitter) zeigt. Somit könnte zumindest ein Diskretisierungsfehler ausgeschlossen werden. Wenn sich der Effekt andererseits nicht in empirischen Experimenten (falls diese überhaupt möglich sind) zeigt, könnte es sich um ein Artefakt des Modells handeln.
Leider lässt sich die begriffliche Spaltung von modellinternen und modellexternen Gesichtspunkten bei der Validitätsanalyse nicht immer strikt trennen. Denn oft müssen explizit artifizielle Annahmen bewusst in das diskretisierte Computermodell eingebaut werden, um ein fehlerfreies Funktionieren des Computers zu gewährleisten. Manchmal sind paradoxerweise solche Eingriffe sogar notwendig, um überhaupt realistische Phä-

nomene zu erzeugen. Davon unabhängig ist, insbesondere bei Simulationsstudien mit sozialwissenschaftlichen Fragestellungen, nicht immer klar, welches theoretische Modell das geeigneteste ist bzw. wie man vorgeht, wenn verschiedene theoretische Modelle mit beschränkter Anwendungsreichweite zusammengebracht werden müssen.

Literatur

Bungartz, Hans-Joachim u. a. (2013). *Modellbildung und Simulation: Eine anwendungsorientierte Einführung*. 2., überarb. Aufl. eXamen.press. Berlin: Springer Spektrum. ISBN: 364237655X.

Deuflhard, Peter und Folkmar Bornemann (2008). *Gewöhnliche Differentialgleichungen: Ordinary differential equations*. 3., durchges. und korrigierte Aufl., [elektronische Ressource]. Bd. / Peter Deuflhard ... ; 2. de Gruyter Lehrbuch. Berlin: de Gruyter. ISBN: 3110203561. DOI: 10.1515/9783110203578. URL: http://www.reference-global.com/doi/book/10.1515/9783110203578.

Hansen, Charles D. und Chris R. Johnson, Hrsg. (2005). *The visualization handbook*. Burlington, MA: Elsevier Butterworth-Heinemann. ISBN: 012387582X. URL: http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=117139.

Holzner, Steven und Judith Muhr (2009). *Differentialgleichungen für Dummies: [differenzierter differenzieren geht nicht]*. 1. Aufl. Für Dummies. Weinheim: Wiley. ISBN: 3527705279. URL: http://sub-hh.ciando.com/book/?bok_id=856839.

Lenhard, Johannes (2015). *Mit allem rechnen - zur Philosophie der Computersimulation: ZTeilweise zugl.: Bielefeld, Univ., Habil.-Schr., 2011*. Ideen & Argumente. Berlin/Boston: de Gruyter. ISBN: 3110401177. URL: http://dx.doi.org/10.1515/9783110401363.

Oberkampf, William L. und Christopher J. Roy (2012). *Verification and validation in scientific computing*. Cambridge: Cambridge Univ. Press. ISBN: 0521113601.

Rolf Rannacher (2005). Heidelberg. URL: http://numerik.iwr.uni-heidelberg.de/~lehre/notes/num0/numerik0.pdf.

Schöning, Uwe (2009). *Theoretische Informatik - kurz gefasst*. 5. Aufl., [Nachdr.] HochschulTaschenbuch. Heidelberg: Spektrum Akad. Verl. ISBN: 3827418240. URL: http://deposit.d-nb.de/cgi-bin/dokserv?id=3030761&prov=M&dok_var=1&dok_ext=htm.

Schumann, Heidrun und Wolfgang Müller (2000). *Visualisierung: Grundlagen und allgemeine methoden*. Berlin und Heidelberg: Springer. ISBN: 978-3-642-57193-0. DOI: 10.1007/978-3-642-57193-0. URL: http://dx.doi.org/10.1007/978-3-642-57193-0.

Abbildungsverzeichnis

1.1 Simulationspipeline 9

2.1 Verlauf der numerischen x^2 und analytischen y_n Lösung 20
2.2 Zeitevolution einer einzelnen schwarzen Zellen mittels Regel 110 23
2.3 Sensitivität von Regel 110 auf den Anfangszustand 24
a Regel 110 mit einem zufälligen Anfangszustand 24
b Regel 110 mit einem anderen zufälligen Anfangszustand 24
2.4 Vergleich der Muster von Weberkegel und Regel 30 24
a Weberkegel Quelle: upload.wikimedia.org/wikipedia/commons/7/7d/Textile_cone.JPG Autor: Richard Ling (wikipedia@rling.com) Lizenz: Attribution-ShareAlike 3.0 Unported https://creativecommons.org/licenses/by-sa/3.0/deed.en 24
b Von Regel 30 erzeugtes Muster 24
2.5 Mit Golly erzeugtes Bild einer Raumschiffkanone 26
2.6 Sugarscape Simulationen 28
a Startzustand nach zufälliger Verteilung der Agenten auf die Karte. 28
b Zustand nach 1520 Simulationsschritten. Die Agenten haben sich an den Punkten mit viel Zucker angesiedelt. Die Vermögensverteilung ist ungleich geworden. 28
2.7 Einheitskreis Quelle: de.wikipedia.org/wiki/Datei:Pi_statistisch.png Autor: Springob in der Wikipedia auf Deutsch Lizenz: Namensnennung - Weitergabe unter gleichen Bedingungen 3.0 Unported https://creativecommons.org/licenses/by-sa/3.0/deed.de 32

3.1 3-dimensionales Vektorfeld Quelle: https://commons.wikimedia.org/wiki/File:Vektorfeld.png Autor: Mth77777 in der Wikipedia auf Deutsch Lizenz: Public Domain 42
3.2 Ausschnitt eines Gitters Quelle: https://commons.wikimedia.org/w/index.php?curid=4500993 Autor: Stefan Friedrich Birkner Lizenz: Attribution-Share Alike 3.0 Unported https://creativecommons.org/licenses/by-sa/3.0/deed.en 53
3.3 Das Riemann-Integral Quelle: https://commons.wikimedia.org/w/index.php?curid=15366521 Autor: Svenlx (Sven Laux) Lizenz: Attribution-Share Alike 3.0 Unported https://creativecommons.org/licenses/by-sa/3.0/deed.en 54

3.4 Numerische Integration . . . 56
3.5 Diskretisierungsfehler . . . 57
3.6 Gesamtfehler der Numerischen Integration . . . 58
3.7 Euler Verfahren . . . 60
3.8 Drei Schritte im Euler Verfahren . . . 61
3.9 Algorithmische Komplexität mit Landausymbolen . . . 64
3.10 Nachzeichnung aus William L. Oberkampf and Christopher J. Roy. Verification and Validation in Scientific Computing. Cambridge University Press, 2010. . . . 68
3.11 Nachzeichnung aus William L. Oberkampf and Christopher J. Roy. Verification and Validation in Scientific Computing. Cambridge University Press, 2010. . . . 70
3.12 Situation nach der Validierung . . . 71
3.13 Validierung mit Gültigkeitsbereich . . . 72
3.14 Overfitting Quelle: https://commons.wikimedia.org/w/index.php?curid=2959742 Autor: Gringer Lizenz: Attribution 3.0 Unported https://creativecommons.org/licenses/by/3.0/deed.en . . . 73
3.15 Kombinierte Kalibrierung und Validierung. . . . 74
3.16 Die einzelnen Schritte der Visualisierungspipeline . . . 82
a Rayleigh-Taylor instability Quelle: https://commons.wikimedia.org/wiki/File:Rayleigh-Taylor_instability.jpg Autor: Lawrence Livermore National Laboratory Lizenz: Public Domain . . . 90
b High Definition Volume Rendering Quelle: https://commons.wikimedia.org/w/index.php?curid=4318591 Autor: Mugab Lizenz: Public Domain . . . 90
c Gravitational waves Quelle: https://commons.wikimedia.org/wiki/File:Gravitywaves.JPG Autoren: The Globus software creators Ian Foster, Carl Kesselman and Steve Tuecke Lizenz: Public Domain . . . 90
d DTI-sagittal-fibers Quelle: https://en.wikipedia.org/wiki/File:DTI-sagittal-fibers.jpg Autor: Thomas Schulz Lizenz: Attribution-Share Alike 3.0 Unported https://creativecommons.org/licenses/by-sa/3.0/deed.en . . . 90
3.18 Voxelgitter Quelle: https://commons.wikimedia.org/w/index.php?curid=3891715 Autor: Thetawave Lizenz: Attribution Share Alike 3.0 Unported https://creativecommons.org/licenses/by-sa/3.0/deed.en . . . 90
a Contour3D Quelle: https://commons.wikimedia.org/w/index.php?curid=11917131 Autor: Shannonbowling Lizenz: Attribution 3.0 Unported https://creativecommons.org/licenses/by/3.0/deed.en . . . 94

b Contour2D Quelle: https://commons.wikimedia.org/w/index.php?curid=18563222 Autor: MHz'as Lizenz: Attribution-Share Alike 3.0 Unported https://creativecommons.org/licenses/by-sa/3.0/deed.en 94
3.20 Links sieht man die aktuelle Zelle, in der die Isofläche angedeutet ist. Nur der Eckpunkt $V3$ liegt innerhalb der von der Isofläche bestimmten Fläche. Rechts im Bild ist stellt das Dreieck eine Approximation an die Isofläche dar. 94
3.21 Die nicht-eindeutige Isoflächen-Approximation im Marching Cube Algorithmus. 95
3.22 Für jeden Pixel wird der Farbwert in Abhängigkeit von Absorption und Emissionseigenschaften aus dem Volumen berechnet. 97